Introduction to the VMS Run-Time Library

Order Number: AA–LA70A–TE

April 1988

This manual provides an overview of the VMS Run-Time Library.

Revision/Update Information:	This document supersedes Sections 1 and 2 of the *VAX/VMS Run-Time Library Routines Reference Manual,* Version 4.4.
Software Version:	VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	CANADA	INTERNATIONAL
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	Digital Equipment Corporation PSG Business Manager c/o Digital§ local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.
In New Hampshire, Alaska, and Hawaii call 603-884-6660.
In Canada call 800-267-6215.
*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).
Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript™ printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

Preface

This manual provides users of the VMS operating system with an overview of the capabilities and functions of the VMS Run-Time Library.

Run-Time Library routines can only be used in programs written in languages that produce native code for the VAX hardware. At present, these languages include VAX MACRO and the following compiled high-level languages:

VAX Ada
VAX BASIC
VAX BLISS-32
VAX C
VAX COBOL
VAX COBOL-74
VAX CORAL
VAX DIBOL
VAX FORTRAN
VAX Pascal
VAX PL/I
VAX RPG
VAX SCAN

Interpreted languages that can also access Run-Time Library routines include VAX DSM and DATATRIEVE.

Intended Audience

This manual is intended for system and application programmers who want to call Run-Time Library routines.

Document Structure

This manual is organized into three chapters as follows:

- Chapter 1 gives an overview of the VMS Run-Time Library.
- Chapter 2 discusses the documentation format used in the reference section of the various Run-Time Library facility manuals.
- Chapter 3 discusses the calling formats used to call Run-Time Library routines.

Associated Documents

The Run-Time Library Routines are documented in a series of reference manuals. This manual provides an overview of the Run-Time Library and a description of how to access its routines. Descriptions of the individual Run-Time Library facilities, along with reference sections describing the individual routines in detail, can be found in the following books:

- The *VMS RTL DECtalk (DTK$) Manual*
- The *VMS RTL Library (LIB$) Manual*
- The *VMS RTL Mathematics (MTH$) Manual*
- The *VMS RTL General Purpose (OTS$) Manual*
- The *VMS RTL Parallel Processing (PPL$) Manual*
- The *VMS RTL Screen Management (SMG$) Manual*
- The *VMS RTL String Manipulation (STR$) Manual*

The VAX Procedure Calling and Condition Handling Standard, which is documented in the *Introduction to VMS System Routines,* contains useful information for anyone who wants to call Run-Time Library routines.

Applications programmers in any language may wish to refer to the *Guide to Creating VMS Modular Procedures* for the Modular Programming Standard and other guidelines.

VAX MACRO programmers will find additional information on calling Run-Time Library routines in the *VAX MACRO and Instruction Set Reference Manual.*

High-level language programmers will find additional information on calling Run-Time Library routines in the language reference manual. Additional information may also be found in the language user's guide provided with your VAX language documentation.

The *Guide to Using VMS Command Procedures* may also be useful.

For a complete list and description of the manuals in the VMS document set, see the *Overview of VMS Documentation.*

Conventions

Convention	Meaning
$\boxed{\text{RET}}$	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

1 Introduction

The VMS Common Run-Time Procedure Library (or simply the Run-Time Library) is a library of prewritten, commonly-used routines that perform a wide variety of operations. These Run-Time Library routines follow the VAX Procedure Calling and Condition Handling Standard and the VMS Modular Programming Standard; hence they are part of the Common Run-Time environment. The Common Run-Time environment lets a program contain routines written in different languages, so that you can call Run-Time Library routines from any VAX language, thus increasing program flexibility.

In this manual, a *routine* is a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name), and optionally an argument list. Procedures and functions are specific types of routines: a *procedure* is a routine that does not return a value, whereas a *function* is a routine that returns a value by assigning that value to the function's identifier.

1.1 Organization of the Run-Time Library

The routines of the VMS Run-Time Library are grouped according to the types of tasks they perform; these groups are referred to as facilities. Each group or facility has an associated prefix that is used in the routine name to identify that routine as a member of a particular facility. Table 1–1 lists all the Run-Time Library facility prefixes and the types of tasks each facility performs.

Table 1–1 Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK$	DECtalk routines that are used to control DIGITAL's DECtalk device
LIB$	Library routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data
MTH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations
OTS$	General purpose routines that perform tasks such as data type conversions as part of a compiler's generated code, and also some mathematical functions
PPL$	Parallel processing routines that simplify subprocess creation, interprocess communication, and resource sharing for parallel applications
SMG$	Screen management routines that are used in designing, composing, and keeping track of complex images on a video screen
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings

The following tables list all the routines available for each of the aforementioned facilities, as well as a brief statement of the routine's function. Table 1–2 lists all the DTK$ facility routines that are used to operate DIGITAL's DECtalk device. For more detailed information on these routines, or on the DTK$ facility in general, refer to the *VMS RTL DECtalk (DTK$) Manual.*

Table 1–2 DTK$ Facility Routines

Routine Name	Function
DTK$ANSWER_PHONE	Wait for the phone to ring and answer
DTK$CHECK_HDWR_STATUS	Check the hardware status
DTK$DIAL_PHONE	Dial the telephone
DTK$HANGUP_PHONE	Hang up the phone
DTK$INITIALIZE	Initialize the DECtalk device
DTK$LOAD_DICTIONARY	Load a word into the DECtalk dictionary
DTK$READ_KEYSTROKE	Read a key entered on the phone keypad

Table 1–2 (Cont.) DTK$ Facility Routines

Routine Name	Function
DTK$READ_STRING	Read a series of keys entered on the phone keypad
DTK$RETURN_LAST_INDEX	Return the last index spoken
DTK$SET_INDEX	Insert an index at the current position
DTK$SET_KEYPAD_MODE	Turn the phone keypad on and off
DTK$SET_LOGGING_MODE	Set the specified logging mode on the DECtalk terminal
DTK$SET_MODE	Set the specified mode on the DECtalk terminal
DTK$SET_SPEECH_MODE	Turn the speech on and off
DTK$SET_TERMINAL_MODE	Set the specified terminal mode on the DECtalk terminal
DTK$SET_VOICE	Set the voice characteristics
DTK$SPEAK_FILE	Speak text from the specified file
DTK$SPEAK_PHONEMIC_TEXT	Speak the specified phonemic text
DTK$SPEAK_TEXT	Speak the specified text
DTK$SPELL_TEXT	Spell out the specified text
DTK$TERMINATE	Terminate the DECtalk device

Table 1–3 lists all of the LIB$ facility routines. For more detailed information on these routines, or on the LIB$ facility in general, refer to the *VMS RTL Library (LIB$) Manual.*

Table 1–3 LIB$ Facility Routines

Routine Name	Function
LIB$ADAWI	Add adjacent word with interlock
LIB$ADD_TIMES	Add two quadword times
LIB$ADDX	Add two multiple-precision binary numbers
LIB$ANALYZE_SDESC	Analyze a string descriptor
LIB$ASN_WTH_MBX	Assign a channel to a mailbox
LIB$AST_IN_PROG	AST in progress
LIB$ATTACH	Attach a terminal to a process
LIB$BBCCI	Test and clear a bit with interlock
LIB$BBSSI	Test and set a bit with interlock
LIB$CALLG	Call a procedure with a general argument list
LIB$CHAR	Transform a byte to the first character of a string
LIB$CONVERT_DATE_STRING	Convert a date string to a quadword

Table 1–3 (Cont.) LIB$ Facility Routines

Routine Name	Function
LIB$CRC	Calculate a Cyclic Redundancy Check (CRC)
LIB$CRC_TABLE	Construct a Cyclic Redundancy Check (CRC) table
LIB$CREATE_DIR	Create a directory
LIB$CREATE_USER_VM_ZONE	Create a user-defined storage zone
LIB$CREATE_VM_ZONE	Create a new storage zone
LIB$CRF_INS_KEY	Insert a key in the cross-reference table
LIB$CRF_INS_REF	Insert a reference to a key in the cross-reference table
LIB$CRF_OUTPUT	Output some cross-reference table information
LIB$CURRENCY	Get the system currency symbol
LIB$CVT_DX_DX	Convert the specified data type
LIB$CVT_FROM_INTERNAL_TIME	Convert internal time to external time
LIB$CVTF_FROM_INTERNAL_TIME	Convert internal time to external time (F-floating value)
LIB$CVT_TO_INTERNAL_TIME	Convert external time to internal time
LIB$CVTF_TO_INTERNAL_TIME	Convert external time to internal time (F-floating value)
LIB$CVT_xTB	Convert numeric text to binary
LIB$CVT_VECTIM	Convert 7-word vector to internal time
LIB$DATE_TIME	Return the date and time as a string
LIB$DAY	Return the day number as a longword integer
LIB$DAY_OF_WEEK	Return the numeric day of the week
LIB$DECODE_FAULT	Decode instruction stream during a fault
LIB$DEC_OVER	Enable or disable decimal overflow detection
LIB$DELETE_FILE	Delete one or more files
LIB$DELETE_LOGICAL	Delete a logical name
LIB$DELETE_SYMBOL	Delete a CLI symbol
LIB$DELETE_VM_ZONE	Delete a virtual memory zone
LIB$DIGIT_SEP	Get the digit separator symbol
LIB$DISABLE_CTRL	Disable CLI interception of control characters
LIB$DO_COMMAND	Execute the specified command
LIB$EDIV	Perform an extended-precision divide

Table 1–3 (Cont.) LIB$ Facility Routines

Routine Name	Function
LIB$EMODF	Perform extended multiply and integerize for F-floating values
LIB$EMODD	Perform extended multiply and integerize for D-floating values
LIB$EMODG	Perform extended multiply and integerize for G-floating values
LIB$EMODH	Perform extended multiply and integerize for H-floating values
LIB$EMUL	Perform an extended-precision multiply
LIB$ENABLE_CTRL	Enable CLI interception of control characters
LIB$ESTABLISH	Establish a condition handler
LIB$EXTV	Extract a field and sign-extend
LIB$EXTZV	Extract a zero-extended field
LIB$FFx	Find the first clear or set bit
LIB$FID_TO_NAME	Convert a device and file ID to a file specification
LIB$FILE_SCAN	Perform a file scan
LIB$FILE_SCAN_END	End of file scan
LIB$FIND_FILE	Find a file
LIB$FIND_FILE_END	End of find file
LIB$FIND_IMAGE_SYMBOL	Merge activate an image symbol
LIB$FIND_VM_ZONE	Find the next valid zone
LIB$FIXUP_FLT	Fix floating reserved operand
LIB$FLT_UNDER	Floating-point underflow detection
LIB$FORMAT_DATE_TIME	Format a date and/or time
LIB$FREE_DATE_TIME_CONTEXT	Free the context used to format a date or time
LIB$FREE_EF	Free an event flag
LIB$FREE_LUN	Free a logical unit number
LIB$FREE_TIMER	Free timer storage
LIB$FREE_VM	Free virtual memory from the program region
LIB$FREE_VM_PAGE	Free a virtual memory page
LIB$GETDVI	Get device/volume information
LIB$GETJPI	Get job/process information
LIB$GETQUI	Get queue information
LIB$GETSYI	Get systemwide information
LIB$GET_COMMAND	Get line from SYS$COMMAND

Table 1–3 (Cont.) LIB$ Facility Routines

Routine Name	Function
LIB$GET_COMMON	Get string from common area
LIB$GET_DATE_FORMAT	Return the user's date input format
LIB$GET_EF	Get an event flag
LIB$GET_FOREIGN	Get foreign command line
LIB$GET_INPUT	Get line from SYS$INPUT
LIB$GET_LUN	Get logical unit number
LIB$GET_MAXIMUM_DATE_LENGTH	Get the maximum possible date/time string length
LIB$GET_SYMBOL	Get the value of a CLI symbol
LIB$GET_USERS_LANGUAGE	Return the user's language choice
LIB$GET_VM	Allocate virtual memory
LIB$GET_VM_PAGE	Get a virtual memory page
LIB$ICHAR	Convert first character of string to integer
LIB$INDEX	Index to relative position of substring
LIB$INIT_DATE_TIME_CONTEXT	Initialize the context used in formatting date/time strings
LIB$INIT_TIMER	Initialize times and counts
LIB$INSERT_TREE	Insert entry in a balanced binary tree
LIB$INSQHI	Insert entry at the head of a queue
LIB$INSQTI	Insert entry at the tail of a queue
LIB$INSV	Insert a variable bit field
LIB$INT_OVER	Integer overflow detection
LIB$LEN	Return the length of a string as a longword
LIB$LOCC	Locate a character
LIB$LOOKUP_KEY	Look up keyword in table
LIB$LOOKUP_TREE	Look up an entry in a balanced binary tree
LIB$LP_LINES	Specify the number of lines on each printer page
LIB$MATCHC	Match characters, return relative position
LIB$MATCH_COND	Match condition values
LIB$MOVC3	Move characters
LIB$MOVC5	Move characters with fill
LIB$MOVTC	Move translated characters
LIB$MOVTUC	Move translated until character
LIB$MULT_DELTA_TIME	Multiply delta time by scalar

Table 1–3 (Cont.) LIB$ Facility Routines

Routine Name	Function
LIB$MULTF_DELTA_TIME	Multiply delta time by F-floating scalar
LIB$PAUSE	Pause program execution
LIB$POLYF	Evaluate polynomials for F-floating values
LIB$POLYD	Evaluate polynomials for D-floating values
LIB$POLYG	Evaluate polynomials for G-floating values
LIB$POLYH	Evaluate polynomials for H-floating values
LIB$PUT_COMMON	Put string into common area
LIB$PUT_OUTPUT	Put line to SYS$OUTPUT
LIB$RADIX_POINT	Radix point symbol
LIB$REMQHI	Remove entry from head of queue
LIB$REMQTI	Remove entry from tail of queue
LIB$RENAME_FILE	Rename one or more files
LIB$RESERVE_EF	Reserve an event flag
LIB$RESET_VM_ZONE	Reset virtual memory zone
LIB$REVERT	Revert to the handler of the procedure activator
LIB$RUN_PROGRAM	Run new program
LIB$SCANC	Scan for characters and return relative position
LIB$SCOPY_DXDX	Copy source string by descriptor to destination
LIB$SCOPY_R_DX	Copy source string by reference to destination
LIB$SET_LOGICAL	Set logical name
LIB$SET_SYMBOL	Set value of a CLI symbol
LIB$SFREE1_DD	Free one or more dynamic strings
LIB$SFREEN_DD	Free *n* dynamic strings
LIB$SGET1_DD	Get one dynamic string
LIB$SHOW_TIMER	Show accumulated times and counts
LIB$SHOW_VM	Show virtual memory statistics
LIB$SHOW_VM_ZONE	Display information about a virtual memory zone
LIB$SIGNAL	Signal exception condition
LIB$SIG_TO_RET	Convert signaled message to a return status

Table 1–3 (Cont.) LIB$ Facility Routines

Routine Name	Function
LIB$SIG_TO_STOP	Convert a signaled condition to a signaled stop
LIB$SIM_TRAP	Simulate floating trap
LIB$SKPC	Skip equal characters
LIB$SPANC	Skip selected characters
LIB$SPAWN	Spawn a subprocess
LIB$STAT_TIMER	Return accumulated time and count statistics
LIB$STAT_VM	Return virtual memory statistics
LIB$STOP	Stop execution and signal the condition
LIB$SUB_TIMES	Subtract two quadword times
LIB$SUBX	Perform multiple-precision binary subtraction
LIB$SYS_ASCTIM	Invoke $ASCTIM to convert binary time to ASCII
LIB$SYS_FAO	Invoke $FAO system service to format output
LIB$SYS_FAOL	Invoke $FAOL system service to format output
LIB$SYS_GETMSG	Invoke $GETMSG system service to get message text
LIB$TPARSE	Implement a table-driven, finite-state parser
LIB$TRA_ASC_EBC	Translate ASCII to EBCDIC
LIB$TRA_EBC_ASC	Translate EBCDIC to ASCII
LIB$TRAVERSE_TREE	Traverse a balanced binary tree
LIB$TRIM_FILESPEC	Fit long file specification into fixed field
LIB$VERIFY_VM_ZONE	Verify a virtual memory zone
LIB$WAIT	Wait a specified period of time

Table 1–4 lists all of the MTH$ facility routines. For more detailed information on these routines, or on the MTH$ facility in general, refer to the *VMS RTL Mathematics (MTH$) Manual.*

Table 1–4 MTH$ Facility Routines

Routine Name	Function
MTH$xACOS	Return arc cosine of angle expressed in radians[1]
MTH$xACOSD	Return arc cosine of angle expressed in degrees[1]
MTH$xASIN	Return arc sine in radians[1]
MTH$xASIND	Return arc sine in degrees[1]
MTH$xATAN	Return arc tangent in radians[1]
MTH$xATAND	Return arc tangent in degrees[1]
MTH$xATAN2	Return arc tangent in radians with two arguments[1]
MTH$xATAND2	Return arc tangent in degrees with two arguments[1]
MTH$xATANH	Return hyperbolic arc tangent[1]
MTH$CxABS	Return complex absolute value[2]
MTH$CCOS	Return complex cosine (F-floating complex value)
MTH$CxCOS	Return complex cosine (D- and G-floating complex values)
MTH$CEXPP	Return complex exponential (F-floating complex value)
MTH$CxEXP	Return complex exponential (D- and G-floating complex values)
MTH$CLOG	Return complex natural logarithm (F-floating complex value)
MTH$CxLOG	Return complex natural logarithm (D- and G-floating complex values)
MTH$CMPLX	Return complex number made from F-floating point values
MTH$xCMPLX	Return complex number made from D- and G-floating values
MTH$CONJG	Return conjugate of a complex number (F-floating point complex value)
MTH$xCONJG	Return conjugate of a complex number (D- and G-floating complex values)
MTH$xCOS	Return cosine of angle expressed in radians[1]
MTH$xCOSD	Return cosine of angle expressed in degrees[1]
MTH$xCOSH	Return hyperbolic cosine[1]
MTH$CSIN	Return complex sine of complex number (F-floating complex value)
MTH$CxSIN	Return complex sine of complex number (D- and G-floating complex values)
MTH$CSQRT	Return complex square root (F-floating point value)
MTH$CxSQRT	Return complex square root (D- and G-floating complex values)
MTH$CVT_x_x	Convert one double-precision value
MTH$CVT_xA_xA	Convert an array of double-precision values
MTH$xEXP	Return an exponential[1]

[1]The routine is valid only for the F-, D-, and G-floating point data types. The corresponding H-floating routine is listed separately with the format MTH$H*routine_name*.

[2]The routine is valid for the three floating-point complex data types: F-, D- and G-floating point complex.

Table 1–4 (Cont.) MTH$ Facility Routines

Routine Name	Function
MTH$HACOS	Return arc cosine in radians (H-floating point value)
MTH$HACOSD	Return arc cosine in degrees (H-floating point value)
MTH$HASIN	Return arc sine in radians (H-floating point value)
MTH$HASIND	Return arc sine in degrees (H-floating point value)
MTH$HATAN	Return arc tangent in radians (H-floating point value)
MTH$HATAND	Return arc tangent in degrees (H-floating point value)
MTH$HATAN2	Return arc tangent in radians (H-floating point) with two arguments
MTH$HATAND2	Return arc tangent in degrees (H-floating point) with two arguments
MTH$HATANH	Return hyperbolic arc tangent (H-floating point value)
MTH$HCOS	Return cosine of angle expressed in radians (H-floating point value)
MTH$HCOSD	Return cosine of angle expressed in degrees (H-floating point value)
MTH$HCOSH	Return hyperbolic cosine (H-floating point value)
MTH$HEXP	Return exponential (H-floating point value)
MTH$HLOG	Return natural logarithm (H-floating point value)
MTH$HLOG2	Return base two logarithm (H-floating point value)
MTH$HLOG10	Return common logarithm (H-floating point value)
MTH$HSIN	Return sine of angle expressed in radians (H-floating point value)
MTH$HSIND	Return sine of angle expressed in degrees (H-floating point value)
MTH$HSINH	Return hyperbolic sine (H-floating point value)
MTH$HSQRT	Return square root (H-floating point value)
MTH$HTAN	Return tangent of angle expressed in radians (H-floating point value)
MTH$HTAND	Return tangent of angle expressed in degrees (H-floating point value)
MTH$HTANH	Compute the hyperbolic tangent (H-floating point value)
MTH$xIMAG	Return imaginary part of a complex number[2]
MTH$xLOG	Return the natural logarithm[1]
MTH$xLOG2	Return base two logarithm[1]
MTH$xLOG10	Return common logarithm[1]
MTH$RANDOM	Generate a random number with uniform distribution
MTH$xREAL	Return real part of a complex number[2]

[1]The routine is valid only for the F-, D-, and G-floating point data types. The corresponding H-floating routine is listed separately with the format MTH$H*routine_name*.

[2]The routine is valid for the three floating-point complex data types: F-, D- and G-floating point complex.

Table 1–4 (Cont.) MTH$ Facility Routines

Routine Name	Function
MTH$xSIN	Return sine of angle expressed in radians[1]
MTH$xSINCOS	Return sine and cosine of angle expressed in radians[3]
MTH$xSINCOSD	Return sine and cosine of angle expressed in degrees[3]
MTH$xSIND	Return sine of angle expressed in degrees[1]
MTH$xSINH	Return hyperbolic sine[1]
MTH$xSQRT	Return square root[1]
MTH$xTAN	Return tangent of angle expressed in radians[1]
MTH$xTAND	Return tangent of angle expressed in degrees[1]
MTH$xTANH	Compute the hyperbolic tangent[1]
MTH$UMAX	Compute the unsigned maximum
MTH$UMIN	Compute the unsigned minimum

[1]The routine is valid only for the F-, D-, and G-floating point data types. The corresponding H-floating routine is listed separately with the format MTH$H*routine_name*.

[3]The routine is valid for the four floating-point data types: F-, D-, G-, and H-floating.

Table 1–5 lists all of the OTS$ facility routines. For more detailed information on these routines, or on the OTS$ facility in general, refer to the *VMS RTL General Purpose (OTS$) Manual*.

Table 1–5 OTS$ Facility Routines

Routine Name	Function
OTS$CNVOUT	Convert D-floating, G-floating, or H-floating to character string
OTS$CVT_L_TB	Convert unsigned integer to binary text
OTS$CVT_L_TI	Convert signed integer to signed integer text
OTS$CVT_L_TL	Convert integer to logical text
OTS$CVT_L_TO	Convert unsigned integer to octal text
OTS$CVT_L_TU	Convert unsigned integer to decimal text
OTS$CVT_L_TZ	Convert integer to hexadecimal text
OTS$CVT_TB_L	Convert binary text to unsigned integer
OTS$CVT_TI_L	Convert signed integer text to integer
OTS$CVT_TL_L	Convert logical text to integer
OTS$CVT_TO_L	Convert octal text to integer
OTS$CVT_TU_L	Convert unsigned decimal text to integer
OTS$CVT_T_z	Convert numeric text to D- or F-floating value
OTS$CVT_T_x	Convert numeric text to G- or H-floating value
OTS$CVT_TZ_L	Convert hexadecimal text to unsigned longword
OTS$DIVCx	Perform complex division

Table 1–5 (Cont.) OTS$ Facility Routines

Routine Name	Function
OTS$DIV_PK_LONG	Perform packed decimal division with long divisor
OTS$DIV_PK_SHORT	Perform packed decimal division with short divisor
OTS$MOVE3	Move data without fill
OTS$MOVE5	Move data with fill
OTS$MULCx	Perform complex multiplication
OTS$POWCxCx	Raise a complex base to a complex floating-point exponent
OTS$POWCxJ	Raise a complex base to a signed longword exponent
OTS$POWDD	Raise a D-floating base to a D-floating exponent
OTS$POWDR	Raise a D-floating base to an F-floating exponent
OTS$POWDJ	Raise a D-floating base to a longword exponent
OTS$POWGx	Raise a G-floating base to a G-floating or longword exponent
OTS$POWGJ	Raise a G-floating base to a longword exponent
OTS$POWHx	Raise an H-floating base to floating-point exponent
OTS$POWHJ	Raise an H-floating base to a longword exponent
OTS$POWII	Raise a word base to a word exponent
OTS$POWHJJ	Raise a longword base to a longword exponent
OTS$POWLULU	Raise an unsigned longword base to an unsigned longword exponent
OTS$POWxLU	Raise a floating-point base to an unsigned longword exponent
OTS$POWRD	Raise an F-floating base to a D-floating exponent
OTS$POWRR	Raise an F-floating base to an F-floating exponent
OTS$POWRJ	Raise an F-floating base to a longword exponent
OTS$SCOPY_DXDX	Copy a source string passed by descriptor to a destination string
OTS$SCOPY_R_DX	Copy a source string passed by reference to a destination string
OTS$SFREE1_DD	Free one dynamic string
OTS$SFREEN_DD	Free *n* dynamic strings
OTS$SGET1_DD	Get one dynamic string

Table 1–6 lists all of the PPL$ facility routines. These routines are used to implement parallel processing applications on VMS systems. For more detailed information on these routines, or on the PPL$ facility in general, refer to the *VMS RTL Parallel Processing (PPL$) Manual.*

Table 1–6 PPL$ Facility Routines

Routine Name	Function
PPL$ADJUST_QUORUM	Adjust the barrier quorum
PPL$AWAIT_EVENT	Await the occurrence of an event
PPL$CREATE_BARRIER	Create a barrier
PPL$CREATE_EVENT	Create an event
PPL$CREATE_SEMAPHORE	Create a semaphore
PPL$CREATE_SHARED_MEMORY	Create shared memory
PPL$CREATE_SPIN_LOCK	Create a spin lock
PPL$CREATE_VM_ZONE	Create a new virtual memory zone
PPL$DECREMENT_SEMAPHORE	Decrement a semaphore to gain access to a resource
PPL$DELETE_SHARED_MEMORY	Delete shared memory
PPL$ENABLE_EVENT_AST	Enable AST notification of an event
PPL$ENABLE_EVENT_SIGNAL	Enable signal notification of an event
PPL$FIND_SYNCH_ELEMENT_ID	Find the synchronization element identifier
PPL$FLUSH_SHARED_MEMORY	Flush shared memory
PPL$GET_INDEX	Get the index of a participant
PPL$INCREMENT_SEMAPHORE	Increment a semaphore to release a resource
PPL$INDEX_TO_PID	Convert a participant-index to a VMS PID
PPL$INITIALIZE	Initialize the PPL$ facility
PPL$PID_TO_INDEX	Convert a VMS PID to a participant-index
PPL$RELEASE_SPIN_LOCK	Release a spin lock
PPL$READ_SEMAPHORE	Read the values associated with a particular semaphore
PPL$SEIZE_SPIN_LOCK	Seize a spin lock
PPL$SET_QUORUM	Set the barrier quorum
PPL$SPAWN	Initiate parallel execution
PPL$STOP	Stop a participant
PPL$TERMINATE	Terminate PPL$ participation
PPL$TRIGGER_EVENT	Trigger an event
PPL$UNIQUE_NAME	Provide a unique name
PPL$WAIT_AT_BARRIER	Synchronize at a barrier

Table 1–7 lists all of the SMG$ facility routines. These routines are used to perform screen management operations. For more detailed information on these routines, or on the SMG$ facility in general, refer to the *VMS RTL Screen Management (SMG$) Manual.*

Table 1–7 SMG$ Facility Routines

Routine Name	Function
SMG$ADD_KEY_DEF	Add a key definition
SMG$BEGIN_DISPLAY_UPDATE	Begin batching of display updates
SMG$BEGIN_PASTEBOARD_UPDATE	Begin batching of pasteboard updates
SMG$CANCEL_INPUT	Cancel input request
SMG$CHANGE_PBD_CHARACTERISTICS	Change pasteboard characteristics
SMG$CHANGE_RENDITION	Change default rendition
SMG$CHANGE_VIEWPORT	Change a viewport associated with a virtual display
SMG$CHANGE_VIRTUAL_DISPLAY	Change a virtual display
SMG$CHECK_FOR_OCCLUSION	Check for occlusion
SMG$CONTROL_MODE	Control mode
SMG$COPY_VIRTUAL_DISPLAY	Copy a virtual display
SMG$CREATE_KEY_TABLE	Create a key table
SMG$CREATE_MENU	Create a menu in a virtual display
SMG$CREATE_PASTEBOARD	Create a pasteboard
SMG$CREATE_SUBPROCESS	Create and initialize a subprocess
SMG$CREATE_VIEWPORT	Create a virtual viewport
SMG$CREATE_VIRTUAL_DISPLAY	Create a virtual display
SMG$CREATE_VIRTUAL_KEYBOARD	Create a virtual keyboard
SMG$CURSOR_COLUMN	Return the cursor column position
SMG$CURSOR_ROW	Return the cursor row position
SMG$DEFINE_KEY	Perform a DEFINE/KEY command
SMG$DEL_TERM_TABLE	Delete a terminal table
SMG$DELETE_CHARS	Delete the specified characters
SMG$DELETE_KEY_DEF	Delete a key definition
SMG$DELETE_LINE	Delete a line
SMG$DELETE_MENU	Delete a menu
SMG$DELETE_PASTEBOARD	Delete a pasteboard
SMG$DELETE_SUBPROCESS	Terminate a subprocess
SMG$DELETE_VIEWPORT	Delete a viewport
SMG$DELETE_VIRTUAL_DISPLAY	Delete a virtual display
SMG$DELETE_VIRTUAL_KEYBOARD	Delete a virtual keyboard
SMG$DISABLE_BROADCAST_TRAPPING	Disable the trapping of broadcast messages
SMG$DISABLE_UNSOLICITED_INPUT	Disable the trapping of unsolicited input
SMG$DRAW_CHAR	Draw the specified character
SMG$DRAW_LINE	Draw a line

Table 1–7 (Cont.) SMG$ Facility Routines

Routine Name	Function
SMG$DRAW_RECTANGLE	Draw a rectangle
SMG$ENABLE_UNSOLICITED_INPUT	Enable the trapping of unsolicited input
SMG$END_DISPLAY_UPDATE	End the batching of display updates
SMG$END_PASTEBOARD_UPDATE	End the batching of pasteboard updates
SMG$ERASE_CHARS	Erase the specified characters
SMG$ERASE_COLUMN	Erase a column from the display
SMG$ERASE_DISPLAY	Erase a virtual display
SMG$ERASE_LINE	Erase a line from the display
SMG$ERASE_PASTEBOARD	Erase a pasteboard
SMG$EXECUTE_COMMAND	Execute the specified command in a subprocess
SMG$FIND_CURSOR_DISPLAY	Find the virtual display that contains the cursor
SMG$FLUSH_BUFFER	Flush the buffer
SMG$GET_BROADCAST_MESSAGE	Get the broadcast message
SMG$GET_CHAR_AT_PHYSICAL_CURSOR	Return the character at the cursor
SMG$GET_DISPLAY_ATTR	Get the display attributes
SMG$GET_KEY_DEF	Get the key definition
SMG$GET_KEYBOARD_ATTRIBUTES	Get the keyboard attributes
SMG$GET_NUMERIC_DATA	Get the numeric data
SMG$GET_PASTEBOARD_ATTRIBUTES	Get the pasteboard attributes
SMG$GET_PASTING_INFO	Get the display pasting information
SMG$GET_TERM_DATA	Get the terminal data
SMG$GET_VIEWPORT_CHAR	Get the characteristics of the display viewport
SMG$HOME_CURSOR	Home the cursor
SMG$INIT_TERM_TABLE	Initialize the terminal table
SMG$INIT_TERM_TABLE_BY_TYPE	Initialize TERMTABLE by VMS terminal type
SMG$INSERT_CHARS	Insert the specified characters
SMG$INSERT_LINE	Insert a line
SMG$INVALIDATE_DISPLAY	Mark a virtual display as invalid
SMG$KEYCODE_TO_NAME	Translate a key code to a key name
SMG$LABEL_BORDER	Label a virtual display border
SMG$LIST_KEY_DEFS	List key definitions
SMG$LIST_PASTING_ORDER	List the display pasting order
SMG$LOAD_KEY_DEFS	Load key definitions

Table 1–7 (Cont.) SMG$ Facility Routines

Routine Name	Function
SMG$LOAD_VIRTUAL_DISPLAY	Load a virtual display from a file
SMG$MOVE_TEXT	Move the specified text
SMG$MOVE_VIRTUAL_DISPLAY	Move a virtual display
SMG$NAME_TO_KEYCODE	Translate a key name to a key code
SMG$PASTE_VIRTUAL_DISPLAY	Paste a virtual display
SMG$POP_VIRTUAL_DISPLAY	Delete a series of virtual displays
SMG$PRINT_PASTEBOARD	Print the pasteboard using a print queue
SMG$PUT_CHARS	Write characters to a virtual display
SMG$PUT_CHARS_HIGHWIDE	Write double-height, double-width characters
SMG$PUT_CHARS_MULTI	Put text with multiple renditions to the display
SMG$PUT_CHARS_WIDE	Write wide characters
SMG$PUT_HELP_TEXT	Output HELP text to a display
SMG$PUT_LINE	Write lines to a virtual display
SMG$PUT_LINE_HIGHWIDE	Write double-height, double-width line
SMG$PUT_LINE_MULTI	Put text with multiple renditions to a display in line mode
SMG$PUT_LINE_WIDE	Write a double-width line
SMG$PUT_PASTEBOARD	Output pasteboard via routine
SMG$PUT_STATUS_LINE	Output a line of text to the hardware status line
SMG$READ_COMPOSED_LINE	Read a composed line
SMG$READ_FROM_DISPLAY	Read text from a display
SMG$READ_KEYSTROKE	Read a single character
SMG$READ_STRING	Read a string
SMG$READ_VERIFY	Read and verify a string
SMG$REMOVE_LINE	Remove a line from a virtual display
SMG$REPAINT_LINE	Repaint one line on the current screen
SMG$REPAINT_SCREEN	Repaint the current screen
SMG$REPASTE_VIRTUAL_DISPLAY	Repaste the virtual display
SMG$REPLACE_INPUT_LINE	Replace the input line
SMG$RESTORE_PHYSICAL_SCREEN	Restore the physical screen
SMG$RETURN_CURSOR_POS	Return the cursor position
SMG$RETURN_INPUT_LINE	Return the input line
SMG$RING_BELL	Ring the terminal bell or buzzer
SMG$SAVE_PHYSICAL_SCREEN	Save the physical screen

Table 1–7 (Cont.) SMG$ Facility Routines

Routine Name	Function
SMG$SAVE_VIRTUAL_DISPLAY	Save the virtual display to a file
SMG$SCROLL_DISPLAY_AREA	Scroll the display area
SMG$SCROLL_VIEWPORT	Scroll a display under a viewport
SMG$SELECT_FROM_MENU	Select an item from a menu
SMG$SET_BROADCAST_TRAPPING	Enable the trapping of broadcast messages
SMG$SET_CURSOR_ABS	Set absolute cursor position
SMG$SET_CURSOR_MODE	Turn the physical cursor on or off
SMG$SET_CURSOR_REL	Set the cursor relative to the current position
SMG$SET_DEFAULT_STATE	Set the default state
SMG$SET_DISPLAY_SCROLL_REGION	Create a display scrolling region
SMG$SET_KEYPAD_MODE	Set the keypad mode
SMG$SET_OUT_OF_BAND_ASTS	Establish an AST routine for out-of-band characters
SMG$SET_PHYSICAL_CURSOR	Set the cursor on the physical screen
SMG$SET_TERM_CHARACTERISTICS	Change the terminal characteristics
SMG$SNAPSHOT	Write a snapshot of the pasteboard
SMG$UNPASTE_VIRTUAL_DISPLAY	Remove the specified virtual display

Table 1–8 lists all of the STR$ facility routines. These routines are used to perform string manipulation operations. For more detailed information on these routines, or on the STR$ facility in general, refer to the *VMS RTL String Manipulation (STR$) Manual.*

Table 1–8 STR$ Facility Routines

Routine Name	Function
STR$ADD	Add two decimal strings
STR$ANALYZE_SDESC	Analyze a string descriptor
STR$APPEND	Append a string
STR$CASE_BLIND_COMPARE	Compare strings without regard to case
STR$COMPARE	Compare two strings
STR$COMPARE_EQL	Compare two strings for equality
STR$COMPARE_MULTI	Compare two strings for equality using the DEC Multinational Character Set
STR$CONCAT	Concatenate two or more strings
STR$COPY_DX	Copy a source string passed by descriptor to a destination string
STR$COPY_R	Copy a source string passed by reference to a destination string

Table 1–8 (Cont.) STR$ Facility Routines

Routine Name	Function
STR$DIVIDE	Divide two decimal strings
STR$DUPL_CHAR	Duplicate character *n* times
STR$ELEMENT	Extract delimited element substring
STR$FIND_FIRST_IN_SET	Find the first character in a set of characters
STR$FIND_FIRST_NOT_IN_SET	Find the first character that does not occur in the set
STR$FIND_FIRST_SUBSTRING	Find the first substring in the input string
STR$FREE1_DX	Free one dynamic string
STR$GET1_DX	Allocate one dynamic string
STR$LEFT	Extract a substring of a string
STR$LEN_EXTR	Extract a substring of a string
STR$MATCH_WILD	Match a wildcard specification
STR$MUL	Multiply two decimal strings
STR$POSITION	Return relative position of a substring
STR$POS_EXTR	Extract a substring of a string
STR$PREFIX	Prefix a string
STR$RECIP	Return the reciprocal of a decimal string
STR$REPLACE	Replace a substring
STR$RIGHT	Extract a substring of a string
STR$ROUND	Round or truncate a decimal string
STR$TRANSLATE	Translate matched characters
STR$TRIM	Trim trailing blanks and tabs
STR$UPCASE	Convert string to all uppercase

1.2 Features of the Run-Time Library

The Run-Time Library provides the following features and capabilities:

- Run-Time Library routines perform a wide range of general utility operations. You can call a Run-Time Library routine from any VAX language instead of writing your own code to perform the operation.

  Routines in the Run-Time Library are part of the VAX Common Run-Time environment; therefore, they can be called from any VAX language. Because they follow the VMS Modular Programming Standard, Run-Time Library routines can be easily incorporated into any program.
- Because many of the routines are shared, they take up less space in memory.
- When new versions of the Run-Time Library are installed, you do not need to revise your calling program, and generally do not need to relink.

- All Run-Time Library routines are fully reentrant unless the description of the facility or the routine specifies otherwise.

  The term *reentrant* means that the routine executes correctly regardless of how many threads of execution are executing at the same time. Currently, reentrancy is supported only when those multiple threads are executing on the same processor. The term *AST-reentrant* means that a routine may be interrupted and reentered from itself or an AST-level thread of execution only. In particular, an AST-reentrant routine may not execute properly if more than one non-AST-level thread of execution is executing the routine at once.

  Because the Run-Time Library routines are reentrant (unless otherwise noted), they can be called from multiple threads of execution. For example, a routine may be called from both an AST-level thread and a non-AST-level thread of an image, as well as from the multiple tasks of an Ada program.

1.3 Linking with the Run-Time Library

Routines in the Run-Time Library execute entirely in the mode of the caller and are intended to be called in user mode. This section explains how to link your program and the Run-Time Library into a single executable unit.

When you link your program, the VMS Linker creates an executable image. If your program includes explicit or implicit calls to the Run-Time Library, the linker automatically searches the following system libraries for the named procedures:

- The system default shareable image library, IMAGELIB.OLB

  The most frequently used portions of the Run-Time Library are contained in this set of shareable images. When you link your program, the linker searches IMAGELIB.OLB to resolve undefined symbols. That is, your program is linked by default with the shareable images in this library to form an executable image.

- The system default object module library, STARLET.OLB

  A portion of the Run-Time Library is contained as object modules in STARLET.OLB. If the linker does not find the shareable image of the routine in IMAGELIB.OLB, it copies the object module of the routine from STARLET.OLB into your program's executable image.

Note that when your program calls a routine that is part of a shareable image, the linker does not copy the routine into your program's executable image, as it does for routines maintained in the object module library.

Using shareable images offers the following advantages:

- Many programs can use the single copy of a shareable image, so each program takes up less space in physical memory and less disk storage.
- More than one program can use a shareable image simultaneously, thus saving memory space.
- When new versions of the Run-Time Library are installed, you do not need to relink the programs that call the shareable Run-Time Library.

[illegible]

1.3 Linking with the Run-Time Library

[illegible]

2 Run-Time Library Documentation Format

Each Run-Time Library routine is documented using a structured format called the routine template. This section discusses the main headings in the routine template, the information that is presented under each heading, and the format used to present the information.

The purpose of this section, therefore, is to explain where to find information and how to read it correctly — not how to use the routines themselves. For information on using Run-Time Library routines, see Chapter 3.

Some main headings in the routine template contain information that requires no further explanation beyond what is given in Table 2–1. However, the following main headings contain information that does require additional discussion, and this discussion takes place in the remaining subsections of this section.

- Format heading
- Returns heading
- Arguments heading
- Condition Values Returned heading

Table 2–1 Main Headings in the Routine Template

Main Heading	Description
Routine name	Required. The routine entry point name appears at the top of the first page. It is usually, though not always, followed by the English name of the routine.
Routine overview	Required. The routine overview appears directly below the routine name. The overview explains, usually in one or two sentences, what the routine does.
Format	Required. The format heading follows the routine overview. The format gives the routine entry point name and the routine argument list. It also specifies whether arguments are required or optional.
Returns	Required. The returns heading follows the routine format. It explains what information is returned by the routine.
Arguments	Required. The arguments heading follows the returns heading. Detailed information about each argument is provided under the arguments heading. If a routine takes no arguments, the word "None" appears.

Table 2–1 (Cont.) Main Headings in the Routine Template

Main Heading	Description
Description	Optional. The description heading follows the arguments heading. The description section contains more detailed information about specific actions taken by the routine: interaction between routine arguments, if any; operation of the routine within the context of VMS; user privileges needed to call the routine, if any; system resources used by the routine; and user quotas that may affect the operation of the routine. Note that any restrictions on the use of the routine are always discussed first in the description section; for example, any required user privileges or necessary system resources are explained first.
Condition Values Returned	Required. The condition values returned section appears following the description section. It lists the condition values (typically status or completion codes) that are returned by the routine.
Example	Optional. The examples heading appears following the condition values returned heading. The example section contains one or more programming examples to illustrate use of the routine. Text explaining the example is most often provided.

2.1 Format Heading

Under the format heading, the following three types of information can be present.

- Procedure call format
- Jump to Subroutine (JSB) format
- Explanatory text

All Run-Time Library routines have a procedure call format, but not all Run-Time Library routines have JSB formats; in fact, most do not. If a routine has a JSB format, it always appears after the routine's procedure call format.

By using the procedure call format, your routine call conforms to the VAX Procedure Calling and Condition Handling Standard. That is, an entry mask is created, registers are saved, and so on.

Use of the JSB call format results in activation of the routine code directly, without the overhead of constructing the entry mask, saving registers, and so on. The JSB call format can be used only by VAX MACRO and VAX BLISS programs.

Explanatory text may appear following one or both of the above formats. This text is present only when needed to clarify the formats.

A procedure call format appears under the format heading as follows:

```
ENTRY_POINT_NAME      arg1  ,arg2  ,[arg3]  ,nullarg  [,arg4]  [,arg5]
```

The preceding format shows the use of the following syntax rules.

- Entry point names

  Entry point names are always shown in uppercase characters.

- Argument names

  Argument names are always shown in lowercase characters.

- Spaces

  One or more spaces are used between the entry point name and the first argument, and between each argument and the next.

- Brackets ([ ])

  Brackets surround optional arguments; in the previous example, **arg3**, **arg4**, and **arg5** are optional arguments because they are surrounded by brackets.

- Commas

  Between arguments, the comma always follows the space. That is, the comma immediately precedes an argument instead of immediately following the previous one. If the argument is optional, the comma may appear inside or outside the brackets. If the optional argument is not the last argument in the list, you must either pass a zero by value or use the comma as a place holder to indicate the place of the omitted argument. If the optional argument is the last argument in the list, you must still include the comma if the comma appears outside the brackets; if the comma appears inside the brackets you can omit the argument entirely.

  For example, **arg3** in the previous example is an optional argument; but because there are other required arguments that follow **arg3** in the list, the comma itself is not optional (since it marks the place of **arg3**); therefore, the comma is not inside the brackets.

  The arguments **arg4** and **arg5** are optional. Because there are no required arguments that follow **arg4** and **arg5** in the list, the commas in front of **arg4** and **arg5** are themselves optional; that is, the commas would not be specified in the call if **arg4** and **arg5** were not specified. Therefore, the commas in front of **arg4** and **arg5** are inside the brackets. Note however that if **arg5** is specified, the comma in front of **arg4** is required whether or not **arg4** is specified.

- Null arguments

  A null argument is a place-holding argument. It is used for either of the following two reasons: (1) to hold a place in the argument list for an argument that has not yet been implemented by DIGITAL but which may be at some future time or (2) to mark the position of an argument that was used in earlier versions of the routine but which is not used in the latest version (upward compatibility is thereby ensured because arguments that follow the null argument in the argument list keep their original positions).

  In the argument list constructed when a procedure is called, both null arguments and omitted optional arguments are represented by longword argument list entries containing the value 0. The programming language syntax required to produce argument list entries containing 0 differs from language to language, so you should refer to your language user's guide for language-specific syntax.

However, in general, the following rule applies to high-level languages: to mark a null argument or to omit an optional argument in the call, specify a comma (,) for each null argument or omitted optional argument.

2.2 Returns Heading

Under the returns heading appears a description of what information, if any, is returned by the routine to the caller. A routine can return information to the caller in various ways. The subsections that follow discuss each possibility and then describe how this returned information is presented under the returns heading.

2.2.1 Condition Values in R0

Most Run-Time Library routines return a condition value in register R0. This condition value contains various kinds of information, but most importantly for the caller, it describes (in bits 0 through 3) the completion status of the operation. Programmers test the condition value to determine if the routine completed successfully, or to determine the cause of the error.

For the purposes of high-level language programmers, the fact that status information is returned by means of a condition value and that it is returned in a VAX register is of little importance because the high-level language programmer receives this status information in the return (or status) variable he or she uses when making the call. The Common Run-Time environment established for high-level languages allows the status information in R0 to be moved automatically to the user's return variable.

Nevertheless, if a routine returns a condition value in R0, the returns heading in the documentation will contain the following information:

VMS Usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

- The "VMS Usage" heading specifies how the data type is interpreted. VMS Usages are discussed in detail further in this chapter.
- The "type" heading specifies the data type of the information returned. Since the data type of a condition value is an unsigned longword, the "type" heading shows "longword (unsigned)".
- The "access" heading specifies the way in which the called routine accesses the object. Since the called routine is returning the condition value, it is writing into this longword; so the "access" heading shows "write only".
- The "mechanism" heading specifies the passing mechanism used by the called routine in returning the condition value. Since the called routine is writing the condition value directly into R0, the mechanism heading shows "by value". (If the called routine had written the address of the condition value into R0, the passing mechanism would have been "by reference".)

Note that if a routine returns a condition value in R0, another main heading in the routine template (Condition Values Returned) describes the possible condition values that the routine can return. This heading is discussed further in this chapter.

2.2.2 Data in Registers R0 Through R11

Some routines return actual data in the VAX registers. The number of registers needed to contain the data depends on the length (or data type) of the information being returned. For example, a Run-Time Library mathematics routine that is returning the cosine of an angle as a G-floating number would use registers R0 and R1 because the length of a G-floating number is two longwords.

If a routine returns actual data in one or more of the registers R0 through R11, the returns heading in the documentation of that routine will contain the following information:

VMS Usage:	yyyyyyyy
type:	xxxxxxxx
access:	write only
mechanism:	by value

The symbol "yyyyyyyy" indicates the VMS usage of the information. In this particular case, the VMS usage would be floating_point.

The symbol "xxxxxxxx" above indicates the data type of the information being returned. For example, for the mathematics routine discussed above, the data type would be G-floating.

Additionally, some explanatory text may be provided following the information about the usage, type, access, and mechanism of the returned value. This text explains other relevant information about what the routine is returning.

It is important to note that, since the routine is returning actual data in the VAX registers, the registers cannot be used to convey completion status information. All routines that return actual data in VAX registers must *signal* a condition value that contains the completion status. If this is the case, the heading reads "Condition Values Signaled". This heading is discussed further in this chapter.

2.3 Arguments Heading

Under the arguments heading appears detailed information about each argument listed in the call format. Arguments are described in the order in which they appear in the call format. If the routine has no arguments, the term "none" appears.

The following format is used to describe each argument.

argument-name

VMS Usage:	VMS-usage-type
type:	argument-data-type
access:	argument-access
mechanism:	argument-passing-mechanism

Additionally, the arguments heading contains at least one paragraph of structured text, followed by other paragraphs of text, as needed.

The following sections discuss each part of the arguments heading separately.

2.3.1 VMS Usage Entry

The VMS usage entry indicates the abstract data structure of the argument. Table 2–2 contains a list of the VMS data structures. Note that most high-level language documentation sets contain a table listing all the VMS usages and the statements required to implement each usage in the appropriate language.

Table 2–2 VMS Data Structures

Data Structure	Definition
access_bit_names	Homogeneous array of 32 quadword descriptors; each descriptor defines the name of one of the 32 bits in an access mask. The first descriptor names bit 0, the second descriptor names bit 1 and so on.
access_mode	Unsigned byte denoting a hardware access mode. This unsigned byte can take four values: 0 specifies kernel mode; 1, executive mode; 2, supervisor mode; and 3, user mode.
address	Unsigned longword denoting the virtual memory address of either data or code, but not of a procedure entry mask (which is of type "procedure").
address_range	Unsigned quadword denoting a range of virtual addresses, which identify an area of memory. The first longword specifies the beginning address in the range; the second longword specifies the ending address in the range.
arg_list	Procedure argument list consisting of one or more longwords. The first longword contains an unsigned integer count of the number of successive, contiguous longwords, each of which is an argument to be passed to a procedure by means of a VAX CALL instruction. The argument list has the following format:

Table 2–2 (Cont.) VMS Data Structures

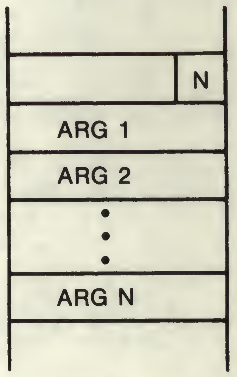
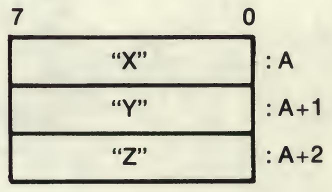
Data Structure	Definition
	 ZK-4204-85
ast_procedure	Unsigned longword integer denoting the entry mask to a procedure to be called at AST level. (Procedures that are not to be called at AST level are of type "procedure".)
boolean	Unsigned longword denoting a Boolean truth value flag. This longword may have only two values: 1 (true) and 0 (false).
byte_signed	This VMS data type is the same as the data type "byte (signed)"in Table 2–3.
byte_unsigned	This VMS data type is the same as the data type "byte integer (unsigned)" in Table 2–3.
channel	Unsigned word integer that is an index to an I/O channel.
char_string	String of from 0 to 65,535 8-bit characters. This VMS data type is the same as the data type "character string" in Table 2–3. The following diagram pictures the character string "XYZ".  ZK-4202-85
complex_number	One of the VAX standard complex floating-point data types. The three complex floating-point numbers are: F-floating complex, D-floating complex, and G-floating complex.

Table 2–2 (Cont.) VMS Data Structures

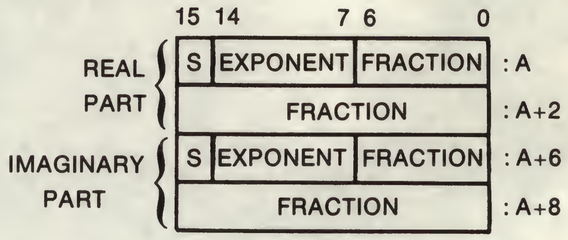
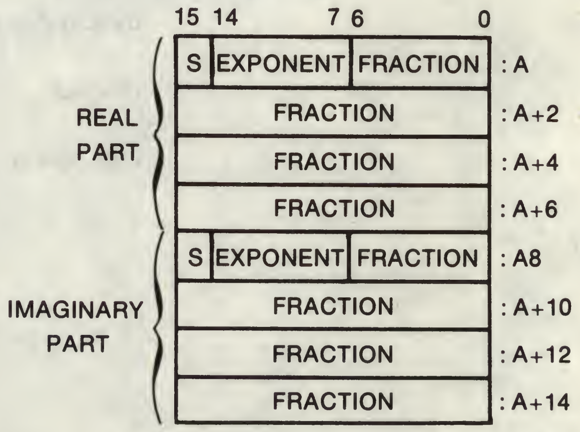
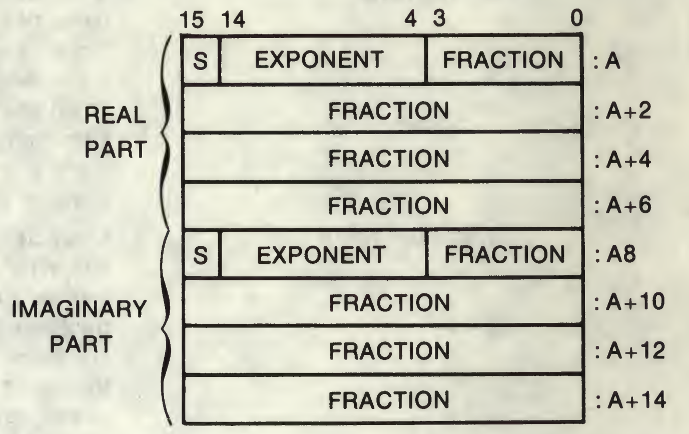
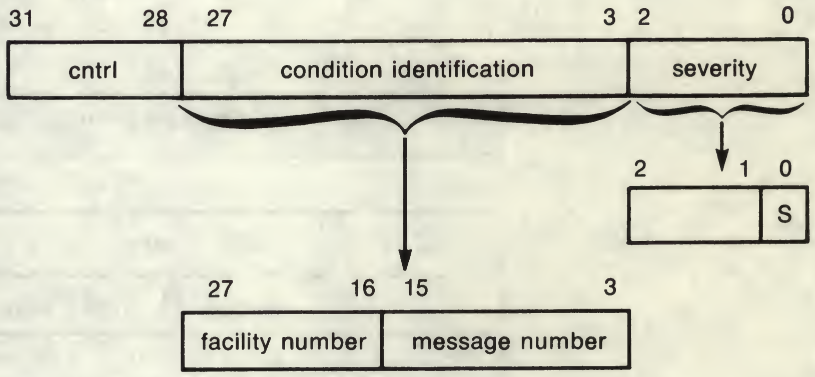
Data Structure	Definition
	An F-floating complex number (r,i) is composed of two F-floating point numbers. The first F-floating point number is the real part (r) of the complex number; the second F-floating point number is the imaginary part (i). The structure of an F-floating complex number is as follows:  15 14 7 6 0 REAL PART IMAGINARY PART S EXPONENT FRACTION :A :A+2 :A+6 :A+8 ZK-4203-85
	A D-floating complex number (r,i) is composed of two D-floating point numbers. The first D-floating point number is the real part (r) of the complex number; the second D-floating point number is the imaginary part (i). The structure of a D-floating complex number is as follows:  15 14 7 6 0 REAL PART IMAGINARY PART S EXPONENT FRACTION :A :A+2 :A+4 :A+6 :A8 :A+10 :A+12 :A+14 ZK-4201-85
	A G-floating complex number (r,i) is composed of two G-floating point numbers. The first G-floating point number is the real part (r) of the complex number; the second G-floating point number is the imaginary part (i). The structure of a G-floating complex number is as follows:

Table 2–2 (Cont.) VMS Data Structures



ZK-4200-85

Data Structure	Definition
cond_value	Unsigned longword integer denoting a condition value (that is, a return status or system condition code), which is typically returned by a procedure in R0. The structure of a condition value is as follows:

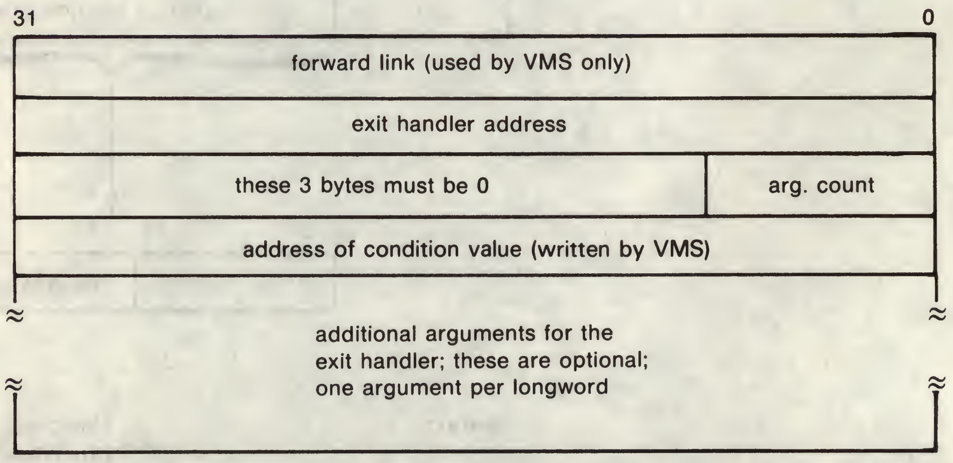


ZK-1795-84

Data Structure	Definition
context	Unsigned longword that is used by a called procedure to maintain position over an iterative sequence of calls. It is usually initialized by the caller, but thereafter manipulated by the called procedure.
date_time	64-bit unsigned, binary integer denoting a date and time as the number of elapsed 100-nanosecond units since 00:00 o'clock, November 17, 1858. This VMS data type is the same as the data type "absolute date and time" in Table 2–3.

Table 2–2 (Cont.) VMS Data Structures

Data Structure	Definition
device_name	Character string denoting the 1- to 9-character name of a device. It can be a logical name, but if it is, it must translate to a valid device name. If the device name contains a colon (:), the colon and the characters past it are ignored. When an underscore (_) precedes the device name string, it indicates that the string is a physical device name.
ef_cluster_name	Character string denoting the 1- to 15-character name of an event flag cluster. It can be a logical name, but if it is, it must translate to a valid event flag cluster name. For more information on how the system translates logical names to global section names see the "Event Flag Services" section of the *Introduction to VMS System Services*.
ef_number	Unsigned longword integer denoting the number of an event flag. Local event flags numbered 32 to 63 are available to your programs.
exit_handler_block	Variable-length structure denoting an exit handler control block. This control block, which describes the exit handler, is depicted in the following diagram.


31
0
forward link (used by VMS only)
exit handler address
these 3 bytes must be 0
arg. count
address of condition value (written by VMS)
additional arguments for the exit handler; these are optional; one argument per longword

ZK-1714-84

Data Structure	Definition
fab	Structure denoting an RMS file access block. A complete description of this structure is contained in the *VMS Record Management Services Manual*.

Table 2–2 (Cont.) VMS Data Structures

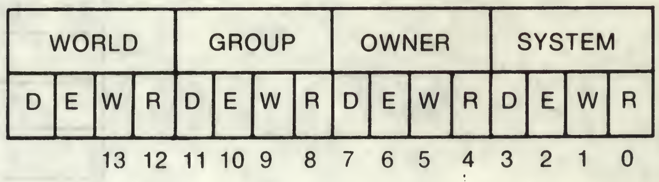
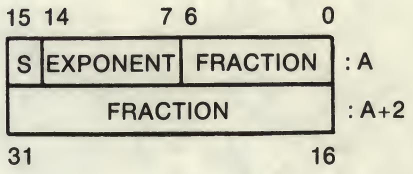
Data Structure	Definition
file_protection	Unsigned word that is a 16-bit mask that specifies file protection. The mask contains four 4-bit fields, each of which specifies the protection to be applied to file access attempts by one of the four categories of user: from the rightmost field to the leftmost field, (1) system users, (2) the file owner, (3) users in the same UIC group as the owner, and (4) all other users (the world). Each field specifies, from the rightmost bit to the leftmost bit: (1) delete access, (2) execute access, (3) write access, (4) read access. Set bits indicate that access is denied. The following diagram depicts the 16-bit file-protection mask.  ZK-1706-84
floating_point	One of the VAX standard floating-point data types. These types are F_floating, D_floating, G_floating, and H_floating. The structure of an F-floating number is as follows:  ZK-4197-85 The structure of a D-floating number is as follows:

Table 2–2 (Cont.) VMS Data Structures

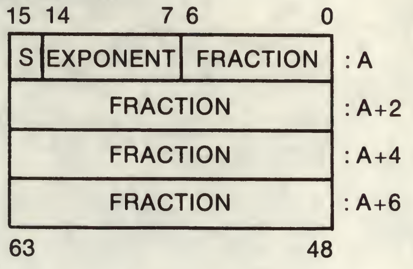
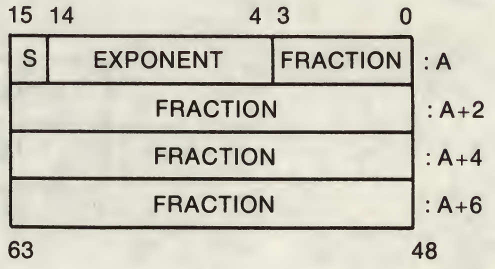
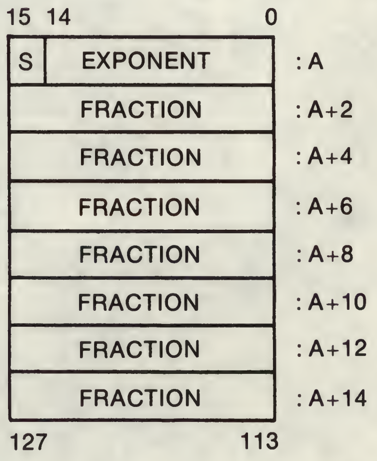
Data Structure	Definition
	 ZK-4198-85 The structure of a G-floating number is as follows:  ZK-4199-85 The structure of an H-floating number is as follows:  ZK-4196-85

Table 2–2 (Cont.) VMS Data Structures

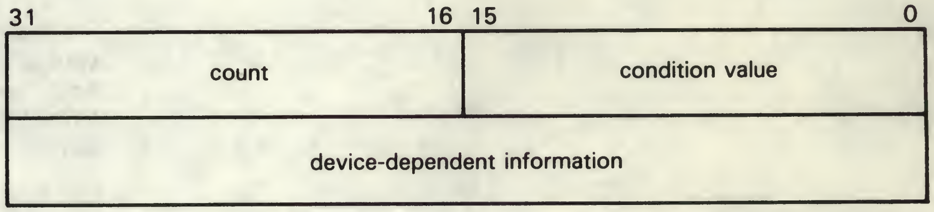
Data Structure	Definition
function_code	Unsigned longword specifying the exact operations a procedure is to perform. This longword has two word-length fields: the first field is a number specifying the major operation; the second field is a mask or bit vector specifying various suboperations within the major operation.
io_status_block	Quadword structure containing information returned by a procedure that completes asychronously. The information returned varies depending on the procedure. The following figure illustrates the format of the information written in the IOSB.  31 16 15 0 count condition value device-dependent information ZK-856-82 The first word contains a condition value indicating the success or failure of the operation. The condition values used are the same as for all returns from system services; for example, SS$_NORMAL indicates successful completion. The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count. For information on specific devices, see the *VMS I/O User's Reference Volume*. The second longword contains device-dependent return information. To ensure successful I/O completion and the integrity of data transfers, the IOSB should be checked following I/O requests, particularly for device-dependent I/O functions. For complete details on how to use the I/O status block, see the *VMS I/O User's Reference Volume*.
item_list_2	Structure that consists of one or more item descriptors and that is terminated by a longword containing 0. Each item descriptor is a 2-longword structure that contains three fields. The following diagram depicts a single item descriptor.

Table 2–2 (Cont.) VMS Data Structures

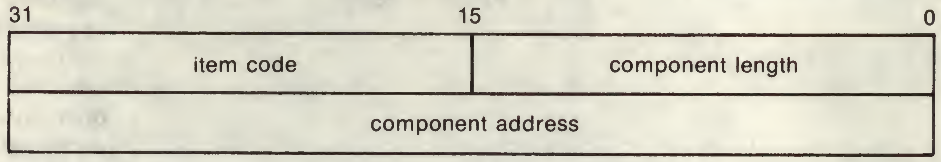
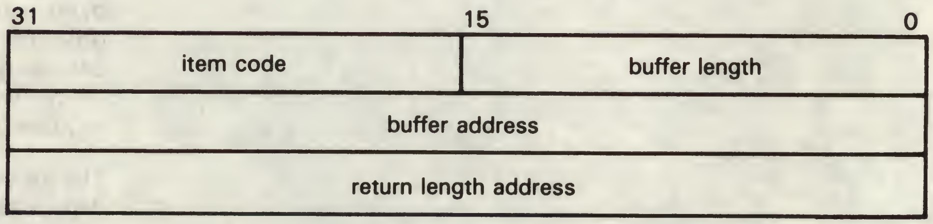
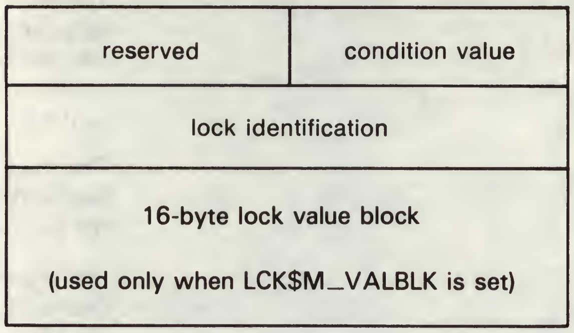
Data Structure	Definition
	 31 15 0 item code component length component address ZK-1709-84 The first field is a word in which the service writes the length (in characters) of the requested component. If the service does not locate the component, it returns the value 0 in this field and in the **component address** field. The second field contains a user-supplied, word-length symbolic code that specifies the component desired. The item codes are defined by the macros that are specific to the service. The third field is a longword in which the service writes the starting address of the component. This address is within the input string itself.
item_list_3	Structure that consists of one or more item descriptors and that is terminated by a longword containing 0. Each item descriptor is a 3-longword structure that contains four fields. The following diagram depicts the format of a single item descriptor.  31 15 0 item code buffer length buffer address return length address ZK-1705-84 The first field is a word containing a user-supplied integer specifying the length (in bytes) of the buffer in which the service writes the information. The length of the buffer needed depends upon the item code specified in the **item code** field of the item descriptor. If the value of **buffer length** is too small, the service truncates the data.

Table 2–2 (Cont.) VMS Data Structures

Data Structure	Definition
	The second field is a word containing a user-supplied symbolic code specifying the item of information that the service is to return. These codes are defined by macros that are specific to the service. The third field is a longword containing the user-supplied address of the buffer in which the service writes the information. The fourth field is a longword containing the user-supplied address of a word in which the service writes the length in bytes of the information it actually returned.
item_quota_list	Structure that consists of one or more quota descriptors and that is terminated by a byte containing a value defined by the symbolic name PQL$_LISTEND. Each quota descriptor consists of a 1-byte quota name followed by an unsigned longword containing the value for that quota.
lock_id	Unsigned longword integer denoting a lock identifier. This lock identifier is assigned by the lock manager facility to a lock when the lock is granted.
lock_status_block	Structure into which the lock manager facility writes status information about a lock. A lock status block always contains at least two longwords: the first word of the first longword contains a condition value; the second word of the first longword is reserved to DIGITAL; and the second longword contains the lock identifier. The lock status block receives the final condition value and the lock identification, and optionally contains a lock value block. When a request is queued, the lock identification is stored in the lock status block even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted. The condition value is placed in the lock status block only when the lock is granted (or when errors occur in granting the lock). The following diagram depicts a lock status block that includes the optional 16-byte lock value block.

Table 2–2 (Cont.) VMS Data Structures



ZK-376-81

Data Structure	Definition
lock_value_block	16-byte block that the lock manager facility includes in a lock status block if the user requests it. The contents of the lock value block are user defined and are not interpreted by the lock manager facility.
logical_name	Character string of from 1 to 255 characters that identifies a logical name or equivalence name to be manipulated by VMS logical name system services. Logical names that denote specific VMS objects have their own VMS types: for example, a logical name identifying a device has the VMS type "device_name".
longword_signed	This VMS data type is the same as the data type "longword integer (signed)" in Table 2–3.
longword_unsigned	This VMS data type is the same as the data type "longword (unsigned)" in Table 2–3.
mask_byte	Unsigned byte wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bit mask".
mask_longword	Unsigned longword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bit mask".
mask_quadword	Unsigned quadword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bit mask".
mask_word	Unsigned word wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bit mask".

Table 2–2 (Cont.) VMS Data Structures

Data Structure	Definition
null_arg	Unsigned longword denoting a "null argument." A "null argument" is an argument whose only purpose is to hold a place in the argument list.
octaword_signed	This VMS data type is the same as the data type "octaword integer (signed)" in Table 2–3.
octaword_unsigned	This VMS data type is the same as the data type "octaword (unsigned)" in Table 2–3.
page_protection	Unsigned longword specifying page protection to be applied by the VAX hardware. Protection values are specified using bits 0 to 3; bits 4 to 31 are ignored. The $PRTDEF macro defines the following symbolic names for the protection codes: (see symbol table below) If the protection is specified as 0, the protection defaults to kernel read-only.
procedure	Unsigned longword denoting the entry mask to a procedure that is not to be called at AST level. (Arguments specifying procedures to be called at AST level have the VMS type "ast_procedure".)

Symbol	Description
PRT$C_NA	No access
PRT$C_KR	Kernel read only
PRT$C_KW	Kernel write
PRT$C_ER	Executive read only
PRT$C_EW	Executive write
PRT$C_SR	Supervisor read only
PRT$C_SW	Supervisor write
PRT$C_UR	User read only
PRT$C_UW	User write
PRT$C_ERKW	Executive read; kernel write
PRT$C_SRKW	Supervisor read; kernel write
PRT$C_SREW	Supervisor read; executive write
PRT$C_URKW	User read; kernel write
PRT$C_UREW	User read; executive write
PRT$C_URSW	User read; supervisor write

Table 2–2 (Cont.) VMS Data Structures

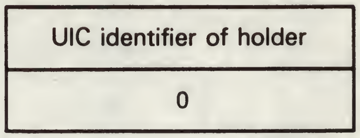
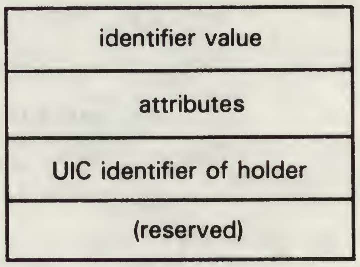
Data Structure	Definition
process_id	Unsigned longword integer denoting a process identifier (PID). This process identifier is assigned by VMS to a process when the process is created.
process_name	Character string, containing 1 to 15 characters, that specifies the name of a process.
quadword_signed	This VMS data type is the same as the data type "quadword integer (signed)" Table 2–3.
quadword_unsigned	This VMS data type is the same as the data type "quadword (unsigned)" in Table 2–3.
rights_holder	Unsigned quadword specifying a user's access rights to a system object. This quadword consists of two fields: the first is an unsigned longword identifier (VMS type "rights_id") and the second is a longword bitmask wherein each bit specifies an access right. Once the identifier record exists in the rights database, you define the holders of that identifier with the $ADD_HOLDER system service. You pass the binary identifier value with the **id** argument; you specify the holder with the **holder** argument, which is the address of a quadword data structure with the following format.  ZK-1903-84 One holder record exists in the rights database for each holder of each identifier. The holder record associates the holder with the identifier, specifies the attributes of the holder, and identifies the UIC identifier of the holder. The format of a holder record is as follows:  identifier value attributes UIC identifier of holder (reserved) ZK-1907-84

Table 2–2 (Cont.) VMS Data Structures

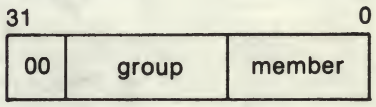
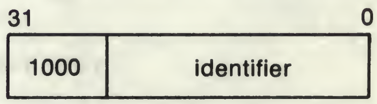
Data Structure	Definition
	The rights database is an indexed file with three keys. The primary key is the identifier value, the secondary key is the holder ID, and the third key is the identifier name. Through the use of the secondary key of the holder ID, all the rights held by a process can be retrieved quickly when LOGINOUT creates the process rights list.
rights_id	Unsigned longword denoting a rights identifier, which identifies an interest group in the context of the VMS security environment. This rights environment may consist of all or part of a user's user identification code (UIC). The basic component of the VMS protection scheme is an identifier. This 32-bit binary value represents various types of agents using the system. The types of agents represented include individual users, groups of users, and environments in which a process is operating. Identifiers have two formats in the rights database: UIC format and ID format. The high-order bits of the identifier value specify the format of the identifier. Two high-order zero bits identify a UIC format identifier; bit 31, set to 1, identifies an ID format identifier. Each UIC identifier is unique and represents a system user. The UIC identifier contains the two high-order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382.  UIC Format ZK-1905-84 Bit 31, set to 1, specifies ID format. Bits 30 through 28 are reserved by DIGITAL. The remaining bits specify the identifier value.  ID Format ZK-1906-84

Table 2–2 (Cont.) VMS Data Structures

Data Structure	Definition
	To the system, an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights database. An identifier name consists of 1 to 31 alphanumeric characters and contains at least one nonnumeric character. An identifier name cannot consist entirely of numeric characters. It can include the characters A through Z, dollar signs ($) and underscores (_), as well as the numbers 0 through 9. Any lowercase characters are automatically converted to uppercase.
rab	Structure denoting an RMS record access block. A complete description of this structure is contained in the *VMS Record Management Services Manual*.
section_id	Unsigned quadword denoting a global section identifier. This identifier specifies the version of a global section and the criteria to be used in matching that global section.
section_name	Character string denoting 1 to 43-character global section name. This character string can be a logical name, but it must translate to a valid global section name. For more information on how the system translates logical names to global section names see the "Memory Management" section of the *Introduction to VMS System Services*.
system_access_id	Unsigned quadword that denotes a system identification value that is to be associated with a rights database.
time_name	Character string specifying a time value in VMS format.
uic	Unsigned longword denoting a user identification code (UIC).
user_arg	Unsigned longword denoting a user-defined argument. This longword is passed to a procedure as an argument, but the contents of the longword are defined and interpreted by the user.
varying_arg	Unsigned longword denoting a variable argument. A variable argument can have variable types, depending on specifications made for other arguments in the call.

Table 2–2 (Cont.) VMS Data Structures

Data Structure	Definition
vector_byte_signed	A homogeneous array whose elements are all signed bytes.
vector_byte_unsigned	A homogeneous array whose elements are all unsigned bytes.
vector_longword_signed	A homogeneous array whose elements are all signed longwords.
vector_longword_unsigned	A homogeneous array whose elements are all unsigned longwords.
vector_quadword_signed	A homogeneous array whose elements are all signed quadwords.
vector_quadword_unsigned	A homogeneous array whose elements are all unsigned quadwords.
vector_word_signed	A homogeneous array whose elements are all signed words.
vector_word_unsigned	A homogeneous array whose elements are all unsigned words.
word_signed	This VMS data type is the same as the data type "word integer (signed)" in Table 2–3.
word_unsigned	This VMS data type is the same as the data type "word (unsigned)" in Table 2–3.

2.3.2 Type Entry

When a calling program passes an argument to a Run-Time Library routine, the routine expects the argument to be of a particular data type. The type entry indicates the expected data type for each argument.

Properly speaking, an argument does not have a data type; rather, the data specified by an argument has a data type. The argument is merely the vehicle for the passing of data to the called routine. Nevertheless, the phrase "argument data type" is frequently used to describe the data type of the data that is specified by the argument.

The following list contains the data types allowed by the VAX Procedure Calling and Condition Handling Standard.

Table 2–3 VAX Data Types

Data Type	Symbolic Code
Absolute date and time	DSC$K_DTYPE_ADT
Byte integer (signed)	DSC$K_DTYPE_B
Bound label value	DSC$K_DTYPE_BLV
Bound procedure value	DSC$K_DTYPE_BPV
Byte (unsigned)	DSC$K_DTYPE_BU
COBOL intermediate temporary	DSC$K_DTYPE_CIT

Table 2–3 (Cont.) VAX Data Types

Data Type	Symbolic Code
D_floating	DSC$K_DTYPE_D
D_floating complex	DSC$K_DTYPE_DC
Descriptor	DSC$K_DTYPE_DSC
F_floating	DSC$K_DTYPE_F
F_floating complex	DSC$K_DTYPE_FC
G_floating	DSC$K_DTYPE_G
G_floating complex	DSC$K_DTYPE_GC
H_floating	DSC$K_DTYPE_H
H_floating complex	DSC$K_DTYPE_HC
Longword integer (signed)	DSC$K_DTYPE_L
Longword (unsigned)	DSC$K_DTYPE_LU
Numeric string, left separate sign	DSC$K_DTYPE_NL
Numeric string, left overpunched sign	DSC$K_DTYPE_NLO
Numeric string, right separate sign	DSC$K_DTYPE_NR
Numeric string, right overpunched sign	DSC$K_DTYPE_NRO
Numeric string, unsigned	DSC$K_DTYPE_NU
Numeric string, zoned sign	DSC$K_DTYPE_NZ
Octaword integer (signed)	DSC$K_DTYPE_O
Octaword (unsigned)	DSC$K_DTYPE_OU
Packed decimal string	DSC$K_DTYPE_P
Quadword integer (signed)	DSC$K_DTYPE_Q
Quadword (unsigned)	DSC$K_DTYPE_QU
Character string	DSC$K_DTYPE_T
Aligned bit string	DSC$K_DTYPE_V
Varying character string	DSC$K_DTYPE_VT
Unaligned bit string	DSC$K_DTYPE_VU
Word integer (signed)	DSC$K_DTYPE_W
Word (unsigned)	DSC$K_DTYPE_WU
Unspecified	DSC$K_DTYPE_Z
Procedure entry mask	DSC$K_DTYPE_ZEM
Sequence of instruction	DSC$K_DTYPE_ZI

2.3.3 Access Entry

The argument access entry describes the way in which the called routine accesses the data specified by the argument. The following three methods of access are the most common.

1. Read only. Data upon which a routine operates, or data needed by the routine to perform its operation, must be read by the called routine. Such data is also called *input* data. When an argument specifies input data, the access entry shows "read only".

   The term "only" is present to indicate that the called routine does not both read and write (that is, "modify") the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.

2. Write only. Data that the called routine returns to the calling routine must be *written* into a location where the calling routine can access it. Such data is also called *output* data. When an argument specifies output data, the access entry shows "write only".

   The term "only" is present to indicate that the called routine does not read the contents of the location either before or after it writes into the location.

3. Modify. When an argument specifies data that is both read and written by the called routine, the access entry shows "modify". In this case, the called routine reads the input data, uses it in its operation, and then overwrites the input data with the results (the output data) of the operation. Thus, when the called routine completes execution, the input data specified by the argument is lost.

The following is a complete list of the access types allowed by the VAX Procedure Calling and Condition Handling Standard.

- Read only
- Write only
- Modify
- Function call (before return)
- JMP after unwind
- Call after stack unwind
- Call without stack unwind

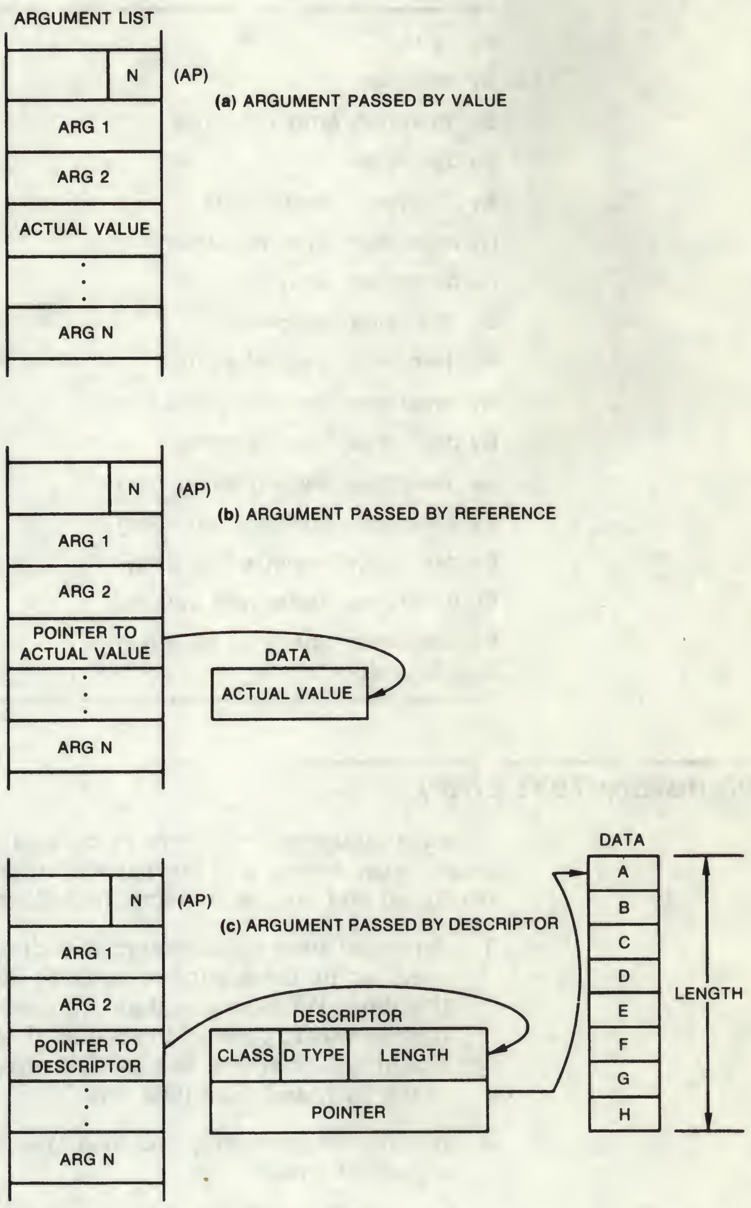
2.3.4 Mechanism Entry

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the argument passing mechanism. There are three types of passing mechanisms.

1. By value. When an argument contains the actual data to be used by the routine, the data is said to be passed to the routine "by value". The argument therefore contains a copy of the actual data. Note that since an actual argument in an argument list is only one longword in length, only data that can be represented in one longword can be passed by value.
2. By reference. When an argument contains the address of the data to be used by the routine, the data is said to be passed "by reference". In this case, the argument is a pointer to the actual data.
3. By descriptor. When an argument contains the address of a descriptor, the data is said to be passed "by descriptor". A descriptor consists of two or more longwords (depending on the type of descriptor used), which describe the location, length, and data type of the data to be used by the called routine. In this case, the argument is a pointer to a descriptor that itself is a pointer to the actual data.

Figure 2–1 illustrates the three passing mechanisms.

Figure 2–1 Routine Argument Passing Mechanisms



Note: ARG 1, ARG 2, and ARG N can be passed by value, by reference, or by descriptor in any of these examples.

:(AP) = argument pointer

N = number of arguments

ZK-1962-84

Table 2–4 contains a list of the passing mechanisms allowed by the VAX Procedure Calling and Condition Handling Standard:

Table 2–4 Passing Mechanisms

Passing Mechanism	Descriptor Code
By value	N/A
By reference	N/A
By reference, array reference	N/A
By descriptor	N/A
By descriptor, fixed-length	DSC$K_CLASS_S
By descriptor, dynamic string	DSC$K_CLASS_D
By descriptor, array	DSC$K_CLASS_A
By descriptor, procedure	DSC$K_CLASS_P
By descriptor, decimal string	DSC$K_CLASS_SD
By descriptor, noncontiguous array	DSC$K_CLASS_NCA
By descriptor, varying string	DSC$K_CLASS_VS
By descriptor, varying string array	DSC$K_CLASS_VSA
By descriptor, unaligned bit string	DSC$K_CLASS_UBS
By descriptor, unaligned bit array	DSC$K_CLASS_UBA
By descriptor, string with bounds	DSC$K_CLASS_SB
By descriptor, unaligned bit string with bounds	DSC$K_CLASS_UBSB

2.3.5 Explanatory Text Entry

For each argument, one or more paragraphs of explanatory text follow the usage, type, access, and mechanism entries. The first paragraph is highly structured and always contains the following items of information.

1. An initial sentence fragment that describes: (1) the nature of the data specified by the argument and (2) the way in which the routine uses this data. For example, if an argument were supplying a number that the routine was to convert to another data type, the initial sentence fragment would be something like the following: "number that is to be converted to the such-and-such data type."
2. A sentence expressing the data type and passing mechanism of the argument data.
   - If the passing mechanism is "by value", this sentence says something like the following: "The **xxx** argument is an unsigned longword containing the such-and-such data."
   - If the passing mechanism is "by reference", this sentence says something like the following: "The **xxx** argument is the address of a *data type* that contains the such-and-such data."

- If the passing mechanism is "by descriptor", this sentence says something like the following: "The **xxx** argument is the address of a descriptor pointing to the such-and-such data."

Additional explanatory paragraphs appear for each argument as needed. For example, some arguments specify complex data consisting of many discrete fields, each of which has a particular purpose and use. In such cases, additional paragraphs provide detailed descriptions of each such field, symbolic names for the fields, if any, and guidance relating to their use.

2.4 Condition Values Returned Heading

A condition value is an unsigned longword that has several uses in the VAX architecture.

- It indicates the success or failure of a called procedure.
- It describes an exception condition when an exception is signaled.
- It identifies system messages.
- It reports program success or failure to the command language level.

The documentation heading "Condition Values Returned" describes the condition values returned by the routine when it completes execution without generating an exception condition. This condition value describes the completion status of the operation.

If a called routine generates an exception condition during execution, the exception condition is *signaled*; the exception condition is then *handled* by a condition handler (either user-supplied or system-supplied). Depending on the nature of the exception condition and the condition handler that handles the exception condition, the called routine will either continue normal execution or terminate abnormally.

If a called Run-Time Library routine executes without generating an exception condition, the called routine either returns a condition value or signals an error condition; a few procedures both return a condition value and signal an error condition. In the documentation of each routine, the method used to return the condition value is indicated in the heading title itself. These heading titles are discussed individually in the subsections that follow.

Under either of these headings, a two-column list gives the symbolic code for each condition value that the routine can return and its accompanying description. This description explains whether the condition value indicates success or failure, and if failure, what user action may have caused the failure and what can be done to correct it. Condition values that indicate success are listed first.

Symbolic codes for condition values are system defined. The symbolic code defined for each condition value equates to a number that is identical to the longword condition value when interpreted as a number. In other words, though the condition value consists of several fields, each of which can be interpreted individually for specific information, the entire longword condition value itself can be interpreted as an unsigned longword integer, which has an equivalent symbolic code.

The following subsections discuss the ways in which a called routine returns condition values.

2.4.1 Condition Values Returned

When the called routine returns a condition value in general register R0, the possible condition values that the routine can return are listed under the "Condition Values Returned" heading. Most routines return a condition value in this way.

2.4.2 Condition Values Signaled

When the called routine signals its condition value (instead of returning it in R0), the possible condition values that the routine can signal are listed under the "Condition Values Signaled" heading.

Routines that signal condition values as a way of indicating the completion status do so because these routines are returning actual data in one or more of the general registers. Since register R0 is used to convey data, it cannot also receive the condition value.

As mentioned, the signaling of condition values occurs whenever a routine generates an exception condition, regardless of how the routine returns its completion status under normal circumstances.

3 How to Call Run-Time Library Procedures

The VAX Procedure Calling and Condition Handling Standard describes the mechanisms used by all VAX languages for invoking routines and passing data between them. In effect, this standard describes the interface between your program and the Run-Time Library routines that your program calls. This chapter describes the basic methods for coding calls to Run-Time Library routines from any VAX language.

In simple terms, when you call a Run-Time Library routine from your program, you must furnish whatever arguments the routine requires. When the routine completes execution, in most cases it returns control to your program. If the routine returns a status code, your program should check the value of the code to determine whether or not the routine completed successfully. If the return status indicates an error, you may want to change the flow of execution of your program to handle the error before returning control to your program.

3.1 Overview

When you log in, the VMS system creates a process that exists until you log out. When you run a program, the system activates an executable image in your process. This image consists of a set of user procedures.

From the Run-Time Library's point of view, *user procedures* are procedures that exist outside the Run-Time Library and that can call Run-Time Library routines. User procedures can additionally call other user procedures that are either supplied by DIGITAL or written by you. According to this definition, then, the Run-Time Library views a VAX native-mode language compiler as a set of user procedures, since the compiler generates code that calls Run-Time Library routines. When you write a program that calls a Run-Time Library routine, the Run-Time Library views your program as a user procedure.

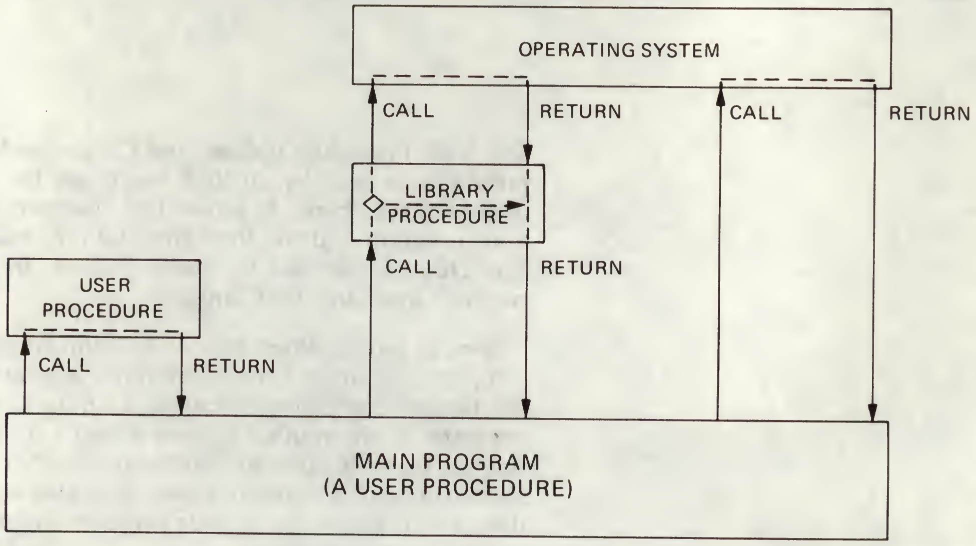
The *main program*, or *main procedure*, is the first user procedure that the system calls after calling a number of initialization procedures. A *user program*, then, consists of the main program and all of the other user procedures that it calls.

Figure 3–1 shows the calling relationships among a main program, other user procedures, library routines, and the VMS operating system. In this figure, CALL indicates that the calling procedures requested some information or action; RETURN indicates that the called procedure returned the information to the calling procedure or performed the action.

Although library routines can always call other library routines or the VMS operating system, they can call user procedures only in the following cases:

- When a user procedure establishes its own condition handler. For example, LIB$SIGNAL operates by searching for and calling user procedures that have been established as condition handlers (see the *VMS RTL Library (LIB$) Manual* for more information).

Figure 3–1 Calling the Run-Time Library



ZK-4262-85

- When a user procedure passes to a routine the address of another procedure that the library will call later. For example, when your program calls LIB$SHOW_TIMER, you can pass the address of an action routine that LIB$SHOW_TIMER will call to process timing statistics.

3.2 Call Formats

Each Run-Time Library routine requires a specific calling sequence. This calling sequence indicates the elements that you must include when calling the routine, and the order of those elements. The form of a calling sequence is explained below.

Call Type

A calling sequence first specifies the type of call being made. A library routine can be invoked by a CALLS or CALLG instruction or by a JSB instruction.

- CALLS - Call Procedure with Stack Argument List instruction
- CALLG - Call Procedure with General Argument List instruction
- JSB - Jump to Subroutine instruction

Note that the following restrictions apply to the different types of calls.

- High-level languages do not differentiate between CALLS and CALLG. They use a CALL statement or a function reference to invoke a Run-Time Library routine.
- MACRO does not differentiate between functions and subroutines in its CALLS and CALLG instructions.

- Only MACRO and BLISS programs can explicitly access the JSB entry points that are provided for some routines in the Run-Time Library. You cannot write a program to access the JSB entry points directly from a high-level language.

Facility Prefix and Routine Name

Each routine is identified by a unique entry point name, consisting of the facility prefix (DTK$, LIB$, MTH$, and so on) and the procedure name (for example, MTH$SIN). Section 3.3 provides more detailed information on entry point naming conventions.

Argument List

Arguments passed to a routine must be listed in your program in the order shown in the format section of the routine description. Each argument has four characteristics: VMS usage, data type, access type, and passing mechanism. These characteristics are described in Chapter 2 of this manual.

Some arguments are optional. Optional arguments are indicated by brackets in the routine descriptions. When your program invokes a Run-Time Library routine using a CALL entry point, you can omit optional arguments at the end of the argument list. If the optional argument is not the last argument in the list, you must either pass a zero by value or use a comma as a place holder to indicate the place of the omitted argument.

Optional arguments apply only to the CALL entry points. JSB entry points do not have optional arguments; all specified registers are used.

For example, the call format for a procedure with two optional arguments is as follows:

```
LIB$GET_INPUT    get-str  [,prompt-str]  [,out-len]
```

A FORTRAN program could include any one of the following calls to this procedure:

```
STAT = LIB$GET_INPUT  (GET_STR,PROMPT,LENGTH)
STAT = LIB$GET_INPUT  (GET_STR,PROMPT)
STAT = LIB$GET_INPUT  (GET_STR,PROMPT,)
STAT = LIB$GET_INPUT  (GET_STR,,LENGTH)
STAT = LIB$GET_INPUT  (GET_STR)
STAT = LIB$GET_INPUT  (GET_STR,)
STAT = LIB$GET_INPUT  (GET_STR,%VAL(0))
```

The following examples illustrate the standard mechanism for calling an external procedure, subroutine, or function in most high-level languages.

BASIC

```
CALL LIB$MOVTC(SRC, FILL, TABLE, DEST)

STATUS = LIB$GET_INPUT(STRING, 'NAME:')
```

BLISS

```
    LOCAL
        MSG_DESC : BLOCK [8,BYTE];

    MSG_DESC [DSC$B_CLASS] = DSC$K_CLASS_S;
    MSG_DESC [DSC$B_DTYPE] = DSC$K_DTYPE_T;
    MSG_DESC [DSC$W_LENGTH] = 5;
    MSG_DESC [DSC$A_POINTER] = MSG;

    STATUS = LIB$PUT_OUTPUT(MSG_DESC);
```

COBOL

```
CALL LIB$MOVTC USING BY DESCRIPTOR
    SRC,
    FILL,
    TABLE,
    DEST,
    GIVING RET-STATUS.
```

FORTRAN

```
CALL LIB$MOVTC(SRC, FILL, TABLE, DEST)

STATUS = LIB$GET_INPUT(STRING, 'NAME:')
```

Pascal

```
RET_STATUS := LIB$MOVTC (SRC, FILL, TABLE, DEST);
```

PL/I

```
CALL LIB$MOVTC(SRC, FILL, TABLE, DEST);

STATUS = LIB$GET_INPUT(STRING, 'NAME:');
```

As these examples show, VAX languages use varying call forms. Each language user's guide gives specific information on calling the Run-Time Library from that language.

In MACRO, a calling sequence takes one of three forms, as illustrated by the following examples:

```
CALLS       #2,G^LIB$GET_INPUT

CALLG       ARGLIST, G^LIB$GET_VM

JSB         G^MTH$SIN_R4
```

3.3 Run-Time Library Naming Conventions

This section explains the naming conventions that the Run-Time Library follows for its entry point names, return status codes, and condition value symbols.

3.3.1 Entry Point Names

Run-Time Library entry points follow the VAX conventions for naming global symbols. A global entry point takes the following general form:

fac$symbol

The elements which make up this format represent the following:

FAC	A 2- or 3-character facility name
SYMBOL	A 1- to 27-character symbol

The facility names are maintained in a systemwide DIGITAL registry. A unique, 12-bit facility number is assigned to each facility name for use in (1) condition value symbols, and (2) condition values in procedure return status codes, signaled conditions, and messages. The high-order bit of this number is 0 for facilities assigned by DIGITAL and 1 for those assigned by Computer Special Services (CSS) and customers. For further information, refer to the VAX Procedure Calling and Condition Handling Standard.

The Run-Time Library facility names are as follows:

DTK$	DECtalk routines
LIB$	Library routines
MTH$	Mathematics routines
OTS$	General purpose routines
PPL$	Parallel processing routines
SMG$	Screen management routines
STR$	String handling routines

3.3.2 JSB Entry Point Names

JSB entry point names follow the naming conventions explained in the previous section, except that they include a suffix indicating the number of the highest register accessed or modified. This helps ensure that the calling program and the called routine will agree on the number of registers that the called routine is going to change.

The following example illustrates the MACRO code that invokes the library routine MTH$SIN_R4 by means of a JSB instruction. As indicated in the JSB entry point name, this routine uses R0 through R4.

```
JSB G^MTH$SIN_R4    ;F-floating sine uses R0 to R4
```

JSB entry points are available only to MACRO and BLISS programs. No VAX high-level language provides a mechanism for accessing JSB entry points explicitly.

3.3.3 Function Return Values

Some Run-Time Library routines return a function value. This is generally a 32-bit value returned in register R0 or a 64-bit value returned in registers R0:R1. When a routine returns a function value, it cannot use R0 and R1 to return a status code. Therefore, such a procedure signals errors rather than returning a status. This is explained in more detail in Chapter 2 of this manual.

In high-level languages, statuses or function return values in R0 appear as the function result.

3.3.4 Facility Return Status and Condition Value Symbols

Library return status and condition value symbols have the following general form:

fac$_abcmnoxyz

The elements which make up this format represent the following:

fac	The 2- or 3-letter facility symbol
abc	The first three letters of the first word of the associated message
mno	The first three letters of the next word
xyz	The first three letters of the third word, if any

Articles and prepositions are not considered significant words in this format. If a significant word is only two letters long, an underscore is used to fill out the third space. Some examples follow. Note that in most facilities the normal or success symbol is an exception to the convention just described.

SS$_NORMAL	Routine successfully completed
LIB$_INSVIRMEM	Insufficient virtual memory
MTH$_FLOOVEMAT	Floating overflow in mathematics library procedure
OTS$_FATINTERR	Fatal internal error in a language-independent support procedure
LIB$_SCRBUFOVF	Screen buffer overflow

3.3.5 Argument Passing Mechanisms

A calling program passes an argument list of longwords to a called routine; each longword in the argument list specifies a single argument. The called routine interprets each argument using one of three standard passing mechanisms: by value, by reference, or by descriptor.

3.3.5.1 Passing Arguments by Value

When your program passes an argument using the *by value* mechanism, the argument list entry contains the actual uninterpreted 32-bit value of the argument. The value mechanism is usually used to pass constants. For example, to pass the constant 100 by value, the calling program puts 100 directly in the argument list.

All VAX high-level languages require you to specify the by-value mechanism explicitly when you call a procedure that accepts an argument by value. FORTRAN, for example, uses the %VAL built-in function, while COBOL uses the BY VALUE qualifier on the CALL [USING] statement.

A FORTRAN program calls a procedure using the by-value mechanism as follows:

```
INCLUDE  '($SSDEF)'
CALL LIB$STOP (%VAL(SS$_INTOVF))
```

A BLISS program calls this procedure as follows:

```
LIB$SIGNAL (SS$_INTOVF)
```

The equivalent MACRO code is as follows:

```
PUSHL    #SS$_INTOVF          ; Push longword by value
CALLS    #1,G^LIB$SIGNAL      ; Call LIB$SIGNAL
```

Note: **Because the Run-Time Library is intended to be called from higher-level languages, most Run-Time Library routines receive arguments by reference, rather than by value, at their CALL entry points.**

3.3.5.2 Passing Arguments by Reference

When your program passes arguments using the *by reference* mechanism, the argument list entry contains the address of the location that contains the value of the argument. For example, if variable *x* is allocated at location 1000, the argument list entry will contain 1000, the address of the value of *x*.

Most languages pass scalar data by reference by default. Therefore, if you simply specify *x* in the CALL statement or function invocation, the language automatically passes the value stored at the location allocated to *x* to the Run-Time Library routine.

A BLISS program calls a procedure using the by-reference mechanism as follows:

```
    LIB$FLT_UNDER (%REF(1))
```

The equivalent MACRO code is as follows:

```
ONE:      .LONG     1                         ; Longword value 1
            .
            .
            .
          PUSHAL    ONE                       ; Push address of longword
          CALLS     #1,G^LIB$FLT_UNDER        ; Call LIB$FLT_UNDER
```

3.3.5.3 Passing Arguments by Descriptor

When a procedure specifies that an argument is passed *by descriptor*, the argument list entry must contain the address of a descriptor for the argument. This mechanism is used to pass more complicated data. A descriptor includes at least the following fields:

Symbol	Description
DSC$W_LENGTH	Length of data (or DSC$W_MAXSTRLEN, maximum length, for varying strings)
DSC$B_DTYPE	Data type
DSC$B_CLASS	Descriptor class code
DSC$A_POINTER	Address at which the data begins

The VAX Procedure Calling and Condition Handling Standard describes these fields in greater detail.

VAX high-level languages include extensions for passing arguments by descriptor. When you specify by descriptor in these languages, the compiler creates the descriptor, defines its fields, and passes the address of the descriptor to the Run-Time Library routine. In some languages, by descriptor is the default passing mechanism for certain types of arguments, such as character strings. For example, the default mechanism for passing strings in VAX BASIC is by descriptor.

```
100    COMMON STRING GREETING = 30
200    CALL LIB$PUT_SCREEN(GREETING)
```

The default mechanism for passing strings in COBOL, however, is by reference. Therefore, when passing a string argument to a Run-Time Library routine from a COBOL program, you must specify BY DESCRIPTOR for the string argument in the CALL statement.

```
CALL LIB$PUT_OUTPUT USING BY DESCRIPTOR GREETING.
```

In MACRO or BLISS, you must define the descriptor's fields explicitly and push its address onto the stack. Following is the MACRO code that corresponds to the previous examples.

```
MSGDSC:     .WORD LEN                   ; DESCRIPTOR:  DSC$W_LENGTH
            .BYTE DSC$K_DTYPE_T         ; DSC$B_DTYPE
            .BYTE DSC$K_CLASS_S         ; DSC$B_CLASS
            .ADDRESS MSG                ; DSC$A_POINTER

MSG:        .ASCII/Hello/               ; String itself
LEN = .-MSG                             ; Define the length of the string

            .ENTRY  EX1,^M<>
            PUSHAQ MSGDSC               ; Push address of descriptor
            CALLS #1,G^LIB$PUT_OUTPUT   ; Output the string
            RET
            .END EX1
```

The equivalent BLISS code looks like this:

```
MODULE BLISS1 (MAIN = BLISS1,       ! Example of calling LIB$PUT_OUTPUT
        IDENT = '1-001',
        ADDRESSING_MODE(EXTERNAL = GENERAL)) =
BEGIN
EXTERNAL ROUTINE
    LIB$STOP,                       ! Stop execution via signaling
    LIB$PUT_OUTPUT;                 ! Put a line to SYS$OUTPUT

FORWARD ROUTINE
    BLISS1 : NOVALUE;

LIBRARY 'SYS$LIBRARY:STARLET.L32';

ROUTINE BLISS1                      ! Routine
        : NOVALUE =

    BEGIN
!+
! Allocate the necessary local storage.
!-
    LOCAL
        STATUS,                             ! Return status
        MSG_DESC : BLOCK [8, BYTE];         ! Message descriptor

    BIND
        MSG = UPLIT('HELLO');

!+
! Initialize the string descriptor.
!-
    MSG_DESC [DSC$B_CLASS] = DSC$K_CLASS_S;
    MSG_DESC [DSC$B_DTYPE] = DSC$K_DTYPE_T;
    MSG_DESC [DSC$W_LENGTH] = 5;
    MSG_DESC [DSC$A_POINTER] = MSG;
!+
! Put out the string.  Test the return status.
! If it is not a success, then signal the RMS error.
!-
    STATUS = LIB$PUT_OUTPUT(MSG_DESC);
    IF NOT .STATUS THEN LIB$STOP(.STATUS);
    END;                    ! End of routine BLISS1
END                         ! End of module BLISS1
ELUDOM
```

3.4 Passing Scalars as Arguments

When you are passing an input scalar value to a Run-Time Library routine, you usually pass it either by reference or by value. You usually pass output scalar arguments by reference to Run-Time Library routines. An output scalar argument is the address of a location where some scalar output of the routine will be stored.

3.5 Passing Arrays as Arguments

Arrays are passed to Run-Time Library routines by reference or by descriptor.

Sometimes, the routine knows the length and dimensions of the array to be received, as in the case of the table passed to LIB$CRC_TABLE. Arrays such as this are normally passed by reference.

In other cases, the routine will actually analyze and operate on the input array. The routine does not necessarily know the length or dimensions of such an input array, so that a descriptor is necessary to provide the information the routine needs to accurately describe the array.

3.6 Passing Strings as Arguments

Strings are passed by descriptor to Run-Time Library routines. The Run-Time Library routine recognizes the following descriptors:

Descriptor	Descriptor Class Code	Numeric Value
Unspecified	DSC$K_CLASS_Z	0
Fixed-length	DSC$K_CLASS_S	1
Dynamic	DSC$K_CLASS_D	2
Array	DSC$K_CLASS_A	4
Scaled decimal	DSC$K_CLASS_SD	9
Noncontiguous array	DSC$K_CLASS_NCA	10
Varying-length	DSC$K_CLASS_VS	11

A Run-Time Library routine writes strings according to the following three types of semantics:

- Fixed length, characterized by an address and a constant length
- Varying length, characterized by an address, a current length, and a maximum length
- Dynamic, characterized by a current address and a current length

3.7 Combinations of Descriptor Class and Data Type

Some combinations of descriptor class and data type are not permitted, either because they are not meaningful or because the VAX Procedure Calling and Condition Handling Standard does not recognize them. Furthermore, the same function may be performed with more than one combination. This section describes the restrictions on the combinations of descriptor classes and data types. These restrictions help to keep procedure interfaces simple by allowing a procedure to accept a limited set of argument formats without sacrificing functional flexibility.

Tables 3–1 to 3–3 show all possible combinations of descriptor classes and data types. For example, Table 3–1 shows that your program can pass an argument to a Run-Time Library routine whose descriptor class is DSC$K_CLASS_A (array descriptor) and whose data type is unsigned byte (DSC$K_DTYPE_BU). The VAX Procedure Calling and Condition Handling Standard does not permit your program to pass an argument whose descriptor class is DSC$K_CLASS_D (decimal string) and whose data type is unsigned byte.

Table 3–1 Atomic Data Types and Descriptor Classes

	DSC$K_CLASS											
	_S = 1	_D = 2	_V = 3	_A = 4	_P = 5	_SD = 9	_NCA = 10	_VS = 11	_VSA = 12	_UBS = 13	_UBA = 14	_BFA = 191
DSC$K_DTYPE_O = 26	Yes	–	–	Yes	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_F = 10	Yes	–	–	Yes	Yes	Yes	Yes	–	–	Yes	Yes	Yes
DSC$K_DTYPE_D = 11	Yes	–	–	Yes	Yes	Yes	Yes	–	–	–	–	Yes
DSC$K_DTYPE_G = 27	Yes	–	–	Yes	Yes	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_H = 28	Yes	–	–	Yes	Yes	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_FC = 12	Yes	–	–	Yes	Yes	–	Yes	–	–	–	–	–
DSC$K_DTYPE_DC = 13	Yes	–	–	Yes	Yes	–	Yes	–	–	–	–	–
DSC$K_DTYPE_GC = 29	Yes	–	–	Yes	Yes	–	Yes	–	–	–	–	–
DSC$K_DTYPE_HC = 30	–	–	–	–	–	–	–	–	–	–	–	–
DSC$K_DTYPE_CIT = 31	Yes	–	–	Yes	–	–	Yes	–	–	–	–	–

Key

Yes	The Calling Standard allows this combination of class and data type.
*	No valid interpretation exists for this combination.
–	The Calling Standard forbids the use of this combination of class and data type. Higher-level languages and their run-time support must conform to this restriction.

ZK-4267/1-85

Table 3–1 (Cont.) Atomic Data Types and Descriptor Classes

	DSC$K_CLASS											
	_S = 1	_D = 2	_V = 3	_A = 4	_P = 5	_SD = 9	_NCA = 10	_VS = 11	_VSA = 12	_UBS = 13	_UBA = 14	_BFA = 191
DSC$K_DTYPE_Z = 0	Yes	–	–	Yes	–	–	Yes	–	–	Yes	Yes	–
DSC$K_DTYPE_BU = 2	Yes	–	–	Yes	Yes	–	Yes	–	–	Yes	Yes	–
DSC$K_DTYPE_WU = 3	Yes	–	–	Yes	–	–	Yes	–	–	Yes	Yes	–
DSC$K_DTYPE_LU = 4	Yes	–	–	Yes	–	–	Yes	–	–	Yes	Yes	–
DSC$K_DTYPE_QU = 5	Yes	–	–	Yes	–	–	Yes	–	–	–	–	–
DSC$K_DTYPE_OU = 25	Yes	–	–	Yes	–	–	Yes	–	–	–	–	–
DSC$K_DTYPE_B = 6	Yes	–	–	Yes	Yes	Yes	Yes	–	–	Yes	Yes	–
DSC$K_DTYPE_W = 7	Yes	–	–	Yes	Yes	Yes	Yes	–	–	Yes	Yes	Yes
DSC$K_DTYPE_L = 8	Yes	–	–	Yes	Yes	Yes	Yes	–	–	Yes	Yes	Yes
DSC$K_DTYPE_Q = 9	Yes	–	–	Yes	–	Yes	Yes	–	–	–	–	–

Key

Yes	The Calling Standard allows this combination of class and data type.
*	No valid interpretation exists for this combination.
–	The Calling Standard forbids the use of this combination of class and data type. Higher-level languages and their run-time support must conform to this restriction.

ZK-4267/2-85

Table 3–2 String Data Types and Descriptor Classes

	DSC$K_CLASS											
	_S = 1	_D = 2	_V = 3	_A = 4	_P = 5	_SD = 9	_NCA = 10	_VS = 11	_VSA = 12	_UBS = 13	_UBA = 14	_BFA = 191
DSC$K_DTYPE_V = 1	Yes	–	–	Yes	–	–	Yes	–	–	Yes	Yes	–
DSC$K_DTYPE_T = 14	Yes	Yes	–	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DSC$K_DTYPE_NU = 15	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_NL = 16	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_NLO = 17	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_NR = 18	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_NLR = 19	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_NZ = 20	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_P = 21	Yes	–	–	–	–	Yes	Yes	–	–	–	–	–
DSC$K_DTYPE_VT = 37	–	–	–	–	–	–	–	Yes	Yes	–	–	–
DSC$K_DTYPE_VU = 34	*	*	*	*	*	*	*	*	*	*	*	*

Key

Yes	The Calling Standard allows this combination of class and data type.
*	No valid interpretation exists for this combination.
–	The Calling Standard forbids the use of this combination of class and data type. Higher-level languages and their run-time support must conform to this restriction.

ZK-4266-85

Table 3–3 Miscellaneous Data Types and Descriptor Classes

	DSC$K_CLASS											
	_S = 1	_D = 2	_V = 3	_A = 4	_P = 5	_SD = 9	_NCA = 10	_VS = 11	_VSA = 12	_UBS = 13	_UBA = 14	_BFA = 191
DSC$K_DTYPE_ZI = 22	Yes	–	–	–	–	*	–	–	–	–	–	–
DSC$K_DTYPE_ZEM = 23	Yes	–	–	–	–	*	–	–	–	–	–	–
DSC$K_DTYPE_DSC = 3 (See Note 3)	–	–	–	Yes	–	*	Yes	–	–	–	–	–
DSC$K_DTYPE_BPV = 32	Yes	–	–	–	–	*	Yes	–	–	–	–	–
DSC$K_DTYPE_BLV = 33	Yes	–	–	–	–	*	Yes	–	–	–	–	–

Key

Yes	The Calling Standard allows this combination of class and data type.
*	No valid interpretation exists for this combination.
–	The Calling Standard forbids the use of this combination of class and data type. Higher-level languages and their run-time support must conform to this restriction.

ZK-4265-85

Note

1 Class types DSC$K_CLASS_PI (6) and DSC$K_CLASS_JI (8) are considered obsolete.

2 Class type DSC$K_CLASS_J (7) is reserved for use by the debugger.

3 A descriptor with data type DSC$K_DTYPE_DSC (24) points to a descriptor that has class DSC$K_CLASS_D (2) and data type DSC$K_DTYPE_T (14). All other class and data type combinations in the target descriptor are reserved for future definition in the standard.

4 DSC$K_CLASS_P is used by VAX FORTRAN. No new VAX languages will use it.

5 The scale factor for DSC$K_CLASS_SD is always a decimal data type. It does not vary with the data type of the data described by the descriptor.

6 For DSC$K_CLASS_UBS and DSC$K_CLASS_UBA, the length field will specify the length of the data field in bits. For example, if the data type is unsigned word (DSC$K_DTYPE_WU), DSC$W_LENGTH equals 16.

3.8 Errors from Run-Time Library Routines

A routine can indicate an error condition to the calling program either by returning a 32-bit condition value in R0 as a completion code or by signaling the error.

A completion code, also called a return status or condition value, is either a success (bit 0 = 1) or error condition value (bit 0 = 0). In an error condition value, the low-order three bits specify the severity of the error. Bits 27 through 16 contain the facility number, and bits 15 through 3 indicate the particular condition. The high-order four bits are control bits. When the called procedure returns a condition value, the calling program can test R0 and choose a recovery path. A general guideline to follow when testing for success or failure is that all success codes have odd values and all error codes have even values.

When the completion code is signaled, the calling program must establish a handler to get control and take appropriate action. (See the *VMS RTL Library (LIB$) Manual* for a description of signaling and condition handling and more information on the condition value.)

3.9 Calling a Library Procedure in MACRO

This section describes how to code MACRO calls to library routines using a CALLS, CALLG, or JSB instruction. The routine descriptions that appear later in this manual describe the entry points for each routine. You can use either a CALLS or a CALLG instruction to invoke a procedure with a CALL entry point. You must use a JSB instruction to invoke a procedure with a JSB entry point. All MACRO calls are explicitly defined.

3.9.1 MACRO Calling Sequence

All Run-Time Library routines have a CALL entry point. Some routines also have a JSB entry point. In MACRO, you invoke a CALL entry point with a CALLS or CALLG instruction. To access a JSB entry point, use a JSB instruction.

Arguments are passed to CALLS and CALLG entry points by a pointer to the argument list. The only difference between the CALLS and CALLG instructions is as follows:

- For CALLS, the calling procedure pushes the argument list onto the stack (in reverse order) before performing the call. The list is automatically removed from the stack upon return.
- For CALLG, the calling program specifies the address of the argument list, which can be anywhere in memory. This list remains in memory upon return.

Both of these instructions have the same effect on the called procedure.

JSB instructions execute faster than CALL instructions. They do not set up a new stack frame, do not change the enabling of hardware traps or faults, and do not preserve the contents of any registers before modifying them. For this reason, you must be careful when invoking a JSB entry point in order to prevent the loss of information stored by the calling program.

Whichever type of call you use, the actual reference to the procedure entry point should use general mode addressing (G^). This ensures that the linker and the image activator will be able to locate the module within the shareable image.

In most cases, you have to tell a library routine where to find input values and store output values. You must select a data type for each argument when you code your program. Most routines accept and return 32-bit arguments.

For input arguments of byte, word, or longword values, you can supply either a constant value, a variable name, or an expression in the Run-Time Library routine call. If you supply a variable name for the argument, the data type of the variable must be as large or larger than the data types that the called procedure requires. If, for example, the called procedure expects a byte in the range 0 to 100, you can use a variable data type of a byte, word, or longword with a value between 0 and 100.

For each output argument, you must declare a variable of exactly the length required to avoid extraneous data. If, for example, the called procedure returns a byte value to a word-length variable, the leftmost eight bits of the variable (15:8) are not overwritten on output. Conversely, if a procedure returns a longword value to a word-length variable, it modifies variables in the next higher word.

3.9.2 CALLS Instruction Example

Before executing a CALLS instruction, you must push the necessary arguments on the stack. Arguments are pushed in reverse order; the last argument listed in the calling sequence is pushed first. The following example shows how a MACRO program calls the procedure that allocates virtual memory in the program region for LIB$GET_VM.

```
        .PSECT DATA     PIC,USR,CON,REL,GBL,NOSHR,NOEXE,RD,WRT,NOVEC
MEM:    .LONG   0                       ; Longword to hold address of
                                        ; allocated memory
LEN:    .LONG   700                     ; Number of bytes to allocate
        .PSECT  CODE    PIC,USR,CON,REL,GBL,SHR,EXE,RD,NOWRT,NOVEC
        .ENTRY   PROG, ^M<>
        PUSHAL  MEM                     ; Push address of longword
                                        ; to receive address of block
        PUSHAL  LEN                     ; Push address of longword
                                        ; containing number of bytes
                                        ; desired
        CALLS   #2, G^LIB$GET_VM        ; Allocate memory
        BLBC    RO, 1$                  ; Branch if memory not available
        RET
1$:     PUSHL   RO                      ; Signal the error
        CALLS   #1, G^LIB$SIGNAL
        RET
        .END     PROG
```

Because the stack grows toward location 0, arguments are pushed onto the stack in reverse order from the order shown in the general format for the routine. Thus, the **base-address** argument, here called START, is pushed first, and then the **number-bytes** argument, called LEN. Upon return from LIB$GET_VM, the calling program tests the return status (**ret-status**), which is returned in R0 and branches to an appropriate error routine if an error occurred.

3.9.3 CALLG Instruction Example

When you use the CALLG instruction, the arguments are set up in any location, and the call includes a reference to the argument list. The following example of a CALLG instruction is equivalent to the preceding CALLS example.

```
ARGLST:
        .LONG        2                  ; Argument list count
        .ADDRESS     LEN                ; Address of longword containing
                                        ; the number of bytes to allocate.
        .ADDRESS     START              ; Address of longword to receive
                                        ; the starting address of the
                                        ; virtual memory allocated.

LEN:    .LONG        20                 ; Number of bytes to allocate
START:  .BLKL        1                  ; Starting address of the virtual
                                        ; memory.

        CALLG   ARGLIST, G^LIB$GET_VM   ; Get virtual memory
        BLBC    R0, ERROR               ; Check for error
        BRB     10$
```

3.9.4 JSB Entry Points

A procedure's JSB entry point name indicates the highest numbered register that the procedure modifies. Thus a procedure with a suffix Rn modifies registers R0 through Rn. (You should always assume that R0 and R1 are modified.) The calling program loads the arguments in the registers before the JSB instruction is executed.

A calling program must use a JSB instruction to invoke a Run-Time Library routine by means of its JSB entry point. You pass arguments to a JSB entry point by placing them in registers in the following manner.

```
NUM:    .FLOAT       0.7853981            ; Constant P1/4
        MOVF         NUM, R0              ; Set up input argument
        JSB          G^MTH$SIN_R4         ; Call F-floating sine procedure
                                          ; Return with value in R0
```

In this example, R4 in the entry point name indicates that MTH$SIN_R4 changes the contents of registers R0 through R4. The routine does not reference or change the contents of registers R5 through R11.

The entry mask of a calling procedure should specify all the registers to be saved if the procedure invokes a JSB routine. This step is necessary because a JSB procedure does not have an entry mask, and thus has no way to specify registers to be saved or restored.

For example, consider program A calling procedure B by means of a CALL entry point.

- Procedure B modifies the contents of R2 through R6, so the contents of these registers are preserved at the time of the CALL.
- Procedure B then invokes procedure C by means of a JSB entry point.

- Procedure C modifies registers R0 through R7.
- When control returns to procedure B, R7 has been modified, but when procedure B passes control back to procedure A, it restores only R2 through R6. Thus the contents of R7 are unpredictable, and program A does not execute as expected. Procedure B should be rewritten so that R2 through R7 are saved in procedure B's entry mask.

A similar problem occurs if the stack is unwound, because unwinding the stack restores the contents of registers for each stack frame as it removes the previous frame. Because a JSB entry point does not create a stack frame, the contents of the registers before the JSB instruction will not be restored unless they were saved in the entry mask of the calling program. You do this by naming the registers to be saved in the calling program's entry mask, so a stack unwind correctly restores all registers from the stack. In the following example, the function Y=PROC(A,B) returns the value Y, where Y = SIN(A)*SIN(B).

```
.ENTRY  PROC, ^M <R2, R3, R4, R5>     ; Save R2:R5
MOVF    @4(AP), R0                    ; R0 = A
JSB     G^MTH$SIN_R4                  ; R0 = SIN(A)
MOVF    R0 , R5                       ; Copy result to register
                                      ; not modified by MTH$SIN
MOVF    @8(AP) , R0                   ; R0 = B
JSB     G^MTH$SIN_R4                  ; R0 = SIN(B)
MULF    R5 , R0                       ; R0 = SIN(A)SIN(B)
RET                                   ; Return
```

3.9.5 Return Status

Your MACRO program can test for errors by examining segments of the 32-bit status code returned by a Run-Time Library routine.

To test for errors, check for a zero in bit zero, using a Branch on Low Bit Set (BLBS) or Branch on Low Bit Clear (BLBC) instruction.

To test for a particular condition value, compare the 32 bits of the return status with the appropriate return status symbol, using a Compare Long (CMPL) instruction or the Run-Time Library routine LIB$MATCH_COND.

There are three ways to define a symbol for the condition value returned by a Run-Time Library routine so that you can compare the value in R0 with a particular error code:

- Using the .EXTRN symbol directive. This causes the assembler to generate an external symbol declaration.
- Using the $facDEF macro call. Calling the $LIBDEF macro, for example, causes the assembler to define all LIB$ condition values.
- By default. The assembler automatically declares the condition value as an external symbol that is defined as a global symbol in the Run-Time Library.

The following example asks for the user's name. It then calls the Run-Time Library routine LIB$GET_INPUT to read the user's response from the terminal. If the string returned is longer than 30 characters (the space allocated to receive the name), LIB$GET_INPUT returns in R0 the condition value equivalent to the error LIB$_INPSTRTRU, 'input string truncated.' This value is defined as a global symbol by default. The example then checks for the specific error by comparing LIB$_INPSTRTRU with the contents of R0. If LIB$_INPSTRTRU is the error returned, the program considers that the routine executed successfully. If any other error occurs, the program handles it as a true error.

```
        $SSDEF                                      ; Define SS$ symbols
        $DSCDEF                                     ; Define DSC$ symbols
        .PSECT  $DATA
PROMPT_D:                                           ; Descriptor for prompt
        .WORD    PROMPT_LEN                         ; Length field
        .BYTE    DSC$K_DTYPE_T                      ; Type field is text
        .BYTE    DSC$K_CLASS_S                      ; Class field is string
        .ADDRESS PROMPT                             ; Address

PROMPT: .ASCII   /NAME: /                           ; String descriptor
PROMPT_LEN = . - PROMPT                             ; Calculate length of
                                                    ; string

STR_LEN = 30                                        ; Use 30-byte string
STRING_D:                                           ; Input string descriptor
        .WORD    STR_LEN                            ; Length field
        .BYTE    DSC$K_DTYPE_T                      ; Type field in text
        .BYTE    DSC$K_CLASS_S                      ; Class field is string
        .ADDRESS STR_AREA                           ; Address
STR_AREA: .BLKB   STR_LEN                           ; Area to receive string

                .PSECT $CODE
                .ENTRY START , ^M<>
                PUSHAQ PROMPT_D                     ; Push address of prompt
                                                    ; descriptor
                PUSHAQ STRING_D                     ; Push address of string
                                                    ; descriptor

                CALLS   #2 , G^LIB$GET_INPUT        ; Get input string
                BLBS    R0 , 10$                    ; Check for success
                CMPL    R0 , #LIB$_INPSTRTRU        ; Error: Was it
                                                    ; truncated string?
                BEQL    10$                         ; No, more serious error
                PUSHL   R0
                CALLS   #1 , G^LIB$SIGNAL

10$:            MOVL    #SS$_NORMAL , R0            ; Success, or name too
                                                    ; long
                RET
               .END     START
```

3.9.6 Function Return Values in MACRO

Function values are generally returned in R0 (32-bit values) or R0:R1 (64-bit) values. A MACRO program can access a function value by referencing R0 or R0:R1 directly. For functions that return a string, the address of the string or the address of its descriptor is returned in R0. If a function needs to return a value larger than 64 bits, it must return the value by using an output argument.

There are some exceptions to these rules:

- JSB entry points in the MTH$ facility return H-floating values in R0:R3.

- One routine, MTH$SINCOS, returns two function values, the sine and the cosine of an angle. Depending on the data type of the function values, the function values are returned in the following registers:

F-floating	R0 through R1
D-floating or G-floating	R0 through R3
H-floating	R0 through R7

As in the case of output arguments, a variable declared to receive the function values must be exactly the same length as the value.

3.10 Calling a Library Routine in BLISS

This section describes how to code BLISS calls to library routines. A called routine can return only one of the following:

- No value.
- A function value (typically, an integer or floating-point number). For example, MTH$SIN returns its result as an F-floating value in R0.
- A return status (typically, a 32-bit condition value) indicating that the routine has either executed successfully or failed. For example, LIB$GET_INPUT returns a return status in R0. If the routine executed successfully, it returns SS$_NORMAL; if not, it returns one of several possible error condition values. BLISS treats the return status like any other value.

3.10.1 BLISS Calling Sequence

Scalar arguments are usually passed to Run-Time Library routines by reference. Thus, when a BLISS program passes a variable, it appears with no preceding period in the procedure-call actual argument list. A constant value can be easily passed using the %REF built-in function.

The following example shows how a BLISS program calls LIB$PUT_OUTPUT. This routine writes a record at the user's terminal.

```
MODULE SHOWTIME(IDENT='1-1' %TITLE'Print time', MAIN=TIMEOUT)=
BEGIN
LIBRARY 'SYS$LIBRARY:STARLET';  ! Defines system services, etc.
MACRO
    DESC[]=%CHARCOUNT(%REMAINING),  ! VAX string descriptor
        UPLIT BYTE(%REMAINING) %;   !   definition
BIND
        FMTDESC=UPLIT( DESC('At the tone, the time will be ',
              %CHAR(7), '!%T' ));
EXTERNAL ROUTINE
        LIB$PUT_OUTPUT: ADDRESSING_MODE(GENERAL);
```

```
ROUTINE TIMEOUT
        =
        BEGIN
        LOCAL
            TIMEBUF: VECTOR[2],             ! 64-bit system time
            MSGBUF: VECTOR[80,BYTE],        ! Output message buffer
            MSGDESC: BLOCK[8,BYTE],         ! Descriptor for message buffer
            RSLT: WORD;                     ! Length of result string

!+
! Initialize the fields of the string descriptor.
!-
        MSGDESC[DSC$B_CLASS]=DSC$K_CLASS_S;
        MSGDESC[DSC$B_DTYPE]=DSC$K_DTYPE_T;
        MSGDESC[DSC$W_LENGTH]=80;
        MSGDESC[DSC$A_POINTER]=MSGBUF[0]

        $GETTIM(TIMADR=TIMEBUF);            ! Get time as 64-bit integer

        $FAOL(CTRSTR=FMTDESC,               ! Format descriptor
            OUTLEN=RSLT,                    ! Output length (only a word!)
            OUTBUF=MSGDESC,                          ! Output buffer desc.
            PRMLST= %REF(TIMEBUF));                  ! Address of 64-bit
                                                     !  time block
        MSGDESC [DSC$W_LENGTH] = .RSLT;              ! Modify output desc.
        RETURN (LIB$PUT_OUTPUT(MSGDESC);             ! Return status
        END;
END
ELUDOM
```

3.10.2 Accessing a Return Status in BLISS

BLISS accesses a function return value or condition value returned in R0 as follows:

```
STATUS = LIB$PUT_OUTPUT(MSG_DESC);
IF NOT .STATUS THEN LIB$STOP(.STATUS);
```

3.10.3 Calling JSB Entry Points from BLISS

Many of the library mathematics routines have JSB entry points. You can efficiently invoke these routines from a BLISS procedure using LINKAGE and EXTERNAL ROUTINE declarations as in the following example.

```
MODULE JSB_LINK (MAIN = MATH_JSB,        ! Example of using JSB linkage
        IDENT = '1-001',
        ADDRESSING_MODE(EXTERNAL = GENERAL)) =
BEGIN
LINKAGE
        LINK_MATH_R4 = JSB (REGISTER = 0;  ! input reg
                            REGISTER = 0): ! output reg
                       NOPRESERVE (0,1,2,3,4)
                       NOTUSED (5,6,7,8,9,10,11);
EXTERNAL ROUTINE
        MTH$SIND_R4 : LINK_MATH_R4;

FORWARD ROUTINE
        MATH_JSB;
```

```
LIBRARY 'SYS$LIBRARY:STARLET.L32';

ROUTINE MATH_JSB =                  ! Routine

    BEGIN
    LOCAL
        INPUT_VALUE : INITIAL (%E'30.0'),
        SIN_VALUE;

!+
! Get the sine of single floating 30 degrees.  The input, 30 degrees,
! is passed in R0, and the answer, is returned in R0.  Registers
! 0 to 4 are modified by MTH$SIND_R4.
!-

    MTH$SIND_R4 (.INPUT_VALUE ; SIN_VALUE);

    RETURN SS$_NORMAL;
    END;                            ! End of routine

END                         ! End of module JSB_LINK
ELUDOM
```

Index

Reader's Comments

Introduction to the VMS Run-Time Library
AA–LA70A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______________________________

____

____

What I like best about this manual is ______________________________

____

____

What I like least about this manual is ______________________________

____

____

I found the following errors in this manual:

Page	Description
____	______________________________
____	______________________________
____	______________________________
____	______________________________
____	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ____________

Company ______________________________ Date ____________

Mailing Address ______________________________

______________________________ Phone ____________

-- Do Not Tear - Fold Here and Tape --

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here --

Cut Along Dotted Line

Reader's Comments

Introduction to the VMS Run-Time Library
AA–LA70A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less ______

What I like best about this manual is ______

What I like least about this manual is ______

I found the following errors in this manual:

Page	Description

Additional comments or suggestions to improve this manual:

I am using **Version** ______ of the software this manual describes.

Name/Title ______ Dept. ______
Company ______ Date ______
Mailing Address ______
______ Phone ______

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line